Protecting Externally Supplied Software in Small Computers

by

Stephen Thomas Kent

September 1980

This research was supported by IBM through discretionary funding made available to the M.I.T. Laboratory for Computer Science.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts
02139

Acknowledgments

A number of individuals have contributed in one way or another to the production of this thesis and/or to my enjoyable, extended stay at the Laboratory for Computer Science. In this small space I can acknowledge only some of those who have aided me in this endeavor. To those who are not included in this brief list I offer my sincere thanks and an apology.

Dr. David Clark has been extremely helpful throughout this ordeal. Through our weekly discussions he provided critical review, encouragement and numerous suggestions that have improved the readability of the final product. Despite his many responsibilities, he always strived to read drafts of chapters quickly and, ofttimes, he succeeded. My readers, Prof. Liba Svobodova and Prof. Fernando Corbato, contributed many helpful suggestions for improving the thesis and I thank them for their perseverance in reading and commenting upon the manuscript.

In the six years I have spent at LCS I have learned much from casual conversations with my fellow students and the lunchtime sub-committee. While working on this thesis I benefited immensely from such conversations, especially those involving Allen Luniewski, Karen Sollins and Dave Reed. I also must thank Wayne Gramlich for his assistance in resolving text formatting problems and Eliot Moss for his help with file transfer problems.

Of course, no list of acknowledgments is complete without mention of the two women in my life: my mother and my wife. Although she has not been involved in production of this thesis, my mother has provided support, counsel and love for almost 30 years and I have benefited immensely from her numerous and varied contributions to my life. I gratefully acknowledge the many important contributions of my wife, Rachel. She has endured my protracted graduate career while pursuing a doctorate of her own, an impressive task in its own right. Even when her own research has not proceeded smoothly, she has encouraged me and commiserated with me. Her meticulous proofreading of this and other documents has been excellent. I could not have written this thesis without her love and understanding.

Finally, I wish to acknowledge the support provided by IBM through discretionary funds made available to the M.I.T. Laboratory for Computer Science.

Protecting Externally Supplied Software in Small Computers

by
Stephen T. Kent

Submitted to the
Department of Electrical Engineering and Computer Science on 22 September 1980
in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

Abstract

The increasing decentralization of computing resources and the proliferation of personal and small business computers create new problems in computer security. One such problem is the protection of *externally supplied software*, i.e., software supplied by other than the users/owners of these small computers. In the case of personal and small business computers, proprietary software serves as the primary example. In distributed systems comprised of autonomously managed nodes, members of the user community may act as vendors of external software in a less formal context. In these contexts dual security requirements arise: vendors require encapsulation of their software to prevent release and to detect modification of information, whereas users require confinement of external software in order to control its access to computer resources. The protection mechanisms developed to support mutually suspicious subsystems in centralized systems are not directly applicable here because of differences in the computing environment, e.g., the need to protect external subsystems from physical attacks mounted by owners of these small computers.

This thesis employs two tools to achieve the security requirements of vendors of external software: tamper-resistant modules (TRMs) and cryptographic techniques. The former provide physical security, i.e., while the TRM is intact it prevents the release or modification of information contained within and breaking into a TRM results in destruction (erasure) of the sensitive information inside. Packaging all of the sensitive components of a computer system (processor and storage) in a single TRM is often impractical, but selected portions of a system can be protected effectively in this fashion. Cryptographic techniques are employed in two ways in

this application: to secure communication among TRMs and to protect information held in physically unprotected storage outside a TRM.

These tools address the problem of encapsulating external software but do not provide the confinement required by users. External software can be confined in two ways: through the use of a secure operating system in conjunction with a TRM supplied by a third-party or by providing separate processors for vendors and users and employing some simple hardware to implement access control for the user. Designing small computer systems incorporating these security features requires careful analysis of a number of options in making tradeoffs among performance, cost, flexibility and security.

Keywords: computer security, protected subsystems, proprietary software, cryptography, personal computers, distributed systems, Data Encryption Standard, public-key cryptography

Table of Contents

Table of Figures

Table of Tables

Chapter One

Introduction

1.1 Motivation

The past several years have witnessed a marked growth in decentralization of computing facilities. Evidence of this trend appears in the proliferation of personal and small business computers and development of distributed computer systems composed of autonomously managed computers. (This last class of computers is the focus of much research and is described in more detail later in this section.) This trend is the result of a number of factors including decreasing hardware costs and a desire to tailor computing resources to individual and organizational needs [7]. Improved protection[1] of information is often listed among the advantages accruing from decentralization of computing resources [33]. In many cases decentralization does make protection easier but at least one security problem that has proven tractable in centralized computers becomes more complex as a result of decentralization. The characterization and solution of this problem is the subject of this thesis.

1.1.1 Protection Problems That are Mitigated by Decentralization

The simplest security mechanisms implemented in centralized computers provide complete isolation of users, perhaps allowing total sharing of some files [29].

---

[1] The terms *protection* and *security* are used throughout this thesis to describe techniques for controlling who may access a computer and the information stored within it; they are not interpreted to encompass threats such as natural disasters.

Decentralized computers implicitly provide isolation since each user is supplied with his own computer. (In fact, some of these computers may support multiple users, but the assumption is that these users are equivalent for protection purposes.) Moreover, the user need not rely on personnel at a central facility to protect his data. Thus simple isolation is better achieved using decentralized computers. More sophisticated protection mechanisms in centralized computers permit users to explicitly control which users may access specific files and what type of access is permitted, e.g., reading or writing. Controlled sharing in decentralized systems is readily accomplished through message transmission over a communication network. Such sharing may simply involve transmitting files between users or may be based on sophisticated schemes for managing distributed databases.

When a network is used to selectively share information, communication security measures are required to protect the transmitted data from disclosure and undetected modification in transit and to securely identify users to one another [16] (providing the basis for access control decisions). These communication security measures may be provided in whole or part by the network or may be exclusively the responsibility of the user, depending on the size and geographic range of the user community, network characteristics and user security requirements. Nonetheless, it is often argued that controlled sharing is better achieved in decentralized systems since such sharing takes place only through message exchanges via a network rather than through shared memory interactions involving an operating system and programs of other users [33].

Some security problems associated with borrowed programs also may be mitigated in decentralized systems. The security concern here is that borrowed software may contain a *Trojan Horse* [3], i.e., the software not only performs its advertised function but also engages in malicious activities. The assumption in this case is that the lender of the software imposes no constraints on its use but that the

borrower wants to control access of the software to his data and he wants to prevent the software from disclosing his data to other users. The protection mechanisms required to control access of borrowed software to user data are the same for both centralized and decentralized systems. Preventing borrowed software from disclosing data to other users is difficult or impossible in centralized systems [29] but may be feasible in decentralized computers, since essentially the only means of *leaking* information to the outside world is via a network. Thus if a borrowed program has no legitimate need for network access, or a very restricted requirement for such access, this problem is easily solved. (Borrowed programs that make significant use of a network as part of their normal function are not more easily confined in decentralized systems.)

1.1.2 Protecting Proprietary Software in Centralized Systems

The preceding discussion indicates that decentralization of computing simplifies the problem of protecting information in many cases. However, the problem of protecting *externally supplied software*, i.e., software supplied by one party (the *vendor*) for restricted use by another party (the *client*), becomes more difficult as a result of decentralization. Proprietary software, sold or rented/leased by a vendor to clients, is the primary example of external software but some distributed systems provide other examples, as described later. Vendors want to restrict clients' access to proprietary software, permitting execution but preventing disclosure of the software. The concern here is that clients may illicitly re-distribute the software or may study the software to extract proprietary algorithms. Vendors also may require a secure accounting capability, including the ability to revoke a client's access to proprietary software (prevent him from executing the software), in support of usage-based and time-based billing policies. In centralized computers proprietary software usually is offered (sold, rented or leased) for execution directly on a client's computer.

However, sometimes proprietary software is made available for a fee through a service bureau (a computer facility that sells computer time and services). The protection measures available to a vendor depend on which way the software is offered.

If proprietary software is executed on a client's computer, a number of *ad hoc* technological protection measures are available to the vendor along with various legal measures (trade-secret licensing, contracts containing non-disclosure clauses, copyrights and patents) [21]. Some vendors do not explicitly attempt to protect their software, believing that various vendor-supplied support services are critical to marketing of the software and that simple theft of the software is not a problem. In many cases only object code is provided, in an effort to conceal the algorithms employed and to preclude maintenance by other than the vendor. Vendors may even include extraneous code or engage in circuitous coding practices to deter a client from extracting the underlying structure of the program or to demonstrate the origin of code in disputes over authorship [8]. Some vendors employ a simple form of cryptographic coding, in which a "bootstrap" program decodes the proprietary software prior to execution. These technological measures usually are not employed to protect databases and the only access revocation mechanism available to vendors is the withholding of enhancements and bug fixes for the software.

If proprietary software is made available to clients through a service bureau, the vendor may take advantage of operating system protection mechanisms that allow clients to execute but not read (copy) or modify the software, e.g., the *ring* protection mechanisms of Multics [30]. These protection mechanisms may be quite sophisticated, allowing the vendor to charge on a per-use or time basis, providing quick revocation of access if a client fails to pay and protecting not only programs but also databases associated with the proprietary software. However, clients using proprietary software at a service bureau facility must trust the facility to safeguard

their information, a problem that usually does not arise if the software is executed on the client's computer. The vendor also must trust the service bureau to act as his agent, protecting his software and properly charging for its use. The client also must pay for computing resources at the service bureau, an unnecessary expense for a client with his own computer facilities. Moreover, clients with their own computer facilities may be further penalized by having to maintain and further process proprietary software input or output at the service bureau or by transporting this data between their facilities and the service bureau.

There is substantial disagreement among vendors as to the effectiveness of either legal or *ad hoc* technological measures for protecting proprietary software. Yet vendors of proprietary software do not seem to be deterred by this situation. In the case of proprietary software executing on client equipment, the client is usually a business or other institution for which there is insufficient financial incentive to attempt to subvert the *ad hoc* technological measures or to risk the possible repercussions of violating the legal protection measures. Thus the lack of sound technological protection mechanisms has not been a serious problem in this context. Proprietary software made available through service bureaus can be protected from clients and it is to the advantage of the service bureaus to provide such protection as they gain financially by forcing users to procure time from the bureaus to run this software. The use of service bureaus as agents for proprietary software also has the advantage that a large number of users can gain access to the software but only a small number of facility personnel need be trusted by the vendor to protect the software. In some instances the vendor of proprietary software may also operate the service bureau, eliminating questions of vendor-service bureau mistrust. Finally, some service bureau users cannot afford their own facilities and thus have no alternative to this way of using proprietary software.

1.1.3 Effects of Decentralization on Protection of External Software

The same types of approaches to protecting externally supplied software are available in the decentralized systems of interest, but the problem may be much more severe in this context. If proprietary software is offered for direct execution on client machines the available technological and legal protection measures may prove inadequate in this marketplace. Some evidence already exists that current owners of personal computers engage in extensive informal trading of proprietary software, in violation of contractual agreements and copyright laws. One supplier of proprietary software for personal computers estimates that as many as 90% of the copies of his software in use were not purchased from him [24]. It may be argued that this alarming statistic is not representative of the market as a whole or that it is not indicative of the fate of sales of such software in the future. In particular, it is probably true that many of the current owners of personal computers are themselves employed in the computer field and are thus more likely to delve into their system hardware and software and engage in these activities than would the average *naive* user.

However, it is difficult to predict the moral climate that will characterize users of such systems and there are other reasons to fear that legal means will be insufficient to protect proprietary software in the personal computer marketplace. The very size of the projected personal computer marketplace and the possibility that a small number of manufacturers may dominate this marketplace (resulting in a large body of software compatible processors) make the emergence of "bootleg" copies of proprietary software a likely event. Even in the case of relatively inexpensive software, violations of copyright seem inevitable if an analogy to phonograph records and home stereo systems can be made. Moreover, the growth of communication networks makes distribution of both legitimate and purloined copies of software easier, further complicating the situation. Vendors could offer

proprietary software through service bureaus, to protect their interests, but this negates many of the features brought about by decentralization, including improved protection for user data. Owners of personal computers may balk at buying time from a service bureau and paying for communications to access these centralized facilities. Thus service bureaus are an inappropriate[2] and perhaps an unacceptable means of offering proprietary software for personal computers.

The preceding comments were directed primarily at personal computers but it seems likely that many of these observations apply to the small business computer market as well. Although the size of this market (in numbers of machines) may not approach that of personal computers, small business computers may proliferate more quickly because their utility is, presumably, readily demonstrable. Small businesses generally have greater purchasing power than individuals and thus more sophisticated (and more costly) proprietary software may appear, increasing the profit potential for vendor and pirate alike. It is hard to project the moral and financial climate that will develop and thus difficult to determine how severe a problem informal trading or sales of bootleged proprietary software may become. Nonetheless, it seems prudent to assume that protection of proprietary software will be as important for small business computers as for personal computers. Again, providing proprietary software through service bureaus is contrary to the decentralization trend and is probably unacceptable in this context. Thus there is a great need for an improved means of protecting proprietary software executed in personal and small business computers.

A slightly different requirement for protection of external software arises in the context of distributed systems comprised of autonomously managed nodes. In these

[2]Only proprietary software that makes use of special facilities not available at the client's computer, e.g., a flatbed plotter or array processing hardware, is best offered through a service bureau.

distributed systems each node (computer) operates under the direction of an independent user, but the users co-operate to provide some services, e.g., distributed databases. Systems of this sort are a topic of current research and there are no extant examples nor experience to draw upon. Nonetheless, one can project protection requirements associated with a form of externally supplied software in this environment, i.e., software produced by a user/vendor at one node for execution at nodes throughout the system. As an example, consider a distributed database that is fully replicated at each node for robustness and for ease of access. The database may contain some information that should not be accessible to some users, even though every node maintains a copy of the database. Thus each user must rely on the database management software to enforce some advertised access control policy at all the nodes.

In the case of a distributed database, the software at each node should prevent unauthorized reading or updating (via messages) by other nodes. It also should prevent unauthorized reading and detect unauthorized update attempts by the node owner. Although it might be possible to *prevent* a node owner from attempting unauthorized updates to the database, such update attempts, if detected, will not affect the integrity of the distributed database as a whole. This is because distributed systems must be prepared to cope with local outages, e.g., a disk crash at a node, without compromising the integrity of the entire database. Thus, if the software at a node determines that a portion of its copy of the database is modified as a result of an attempted unauthorized update by the node owner, the software will treat that portion as damaged, and not affect other nodes.

In general, in these distributed systems, it seems desirable to be able to install software at a node (with the permission of the node owner) which can be protected from unauthorized disclosure and undetected modification. The availability of mechanisms that provide such protection for external software enhances

significantly the flexibility of distributed systems composed of autonomous nodes. For example, distributed instances of extended type managers [33] could be created at one node and made available throughout the system in a secure fashion. Objects could be created at one node and transmitted to other nodes with the assurance that only the type manager for the objects would be able to examine and "appropriately" modify the representation of the objects. Although a number of other mechanisms are required to support this sort of object migration, the ability to protect copies of a distributed type manager at each node (from attacks by the node owner) is central to the concept. These security requirements cannot be met by the use of a centralized computing facility without seriously compromising the distributed nature of these systems.

The preceding discussion has shown how the need for protection of externally supplied software in the decentralized systems of interest differs, in some respects, from the need for such protection in centralized systems. First, the legal and *ad hoc* technical measures employed to protect proprietary software executing on client computers may be inadequate in the case of decentralized systems. Second, use of proprietary software offered through service bureaus negates many of the advantages of decentralization and thus may be unacceptable to users of personal and small business computers. Finally, distributed systems composed of autonomous nodes present new examples of externally supplied software which, if they can be adequately protected, could significantly enhance the flexibility of such systems. This suggests that improved technological measures for protecting externally supplied software for execution on client computers are required for the decentralized computer systems described in this section. The next section provides a more precise statement of the problem and establishes criteria by which proposed solutions will be evaluated.

1.2 Problem Definition and Solution Criteria

The preceding section identified two examples of externally supplied software that require protection in the decentralized systems environment: proprietary programs for personal or small business computers and distributed applications software for certain types of distributed systems. This section examines in greater detail the security requirements associated with these examples and abstracts from them a general statement of the problem to be solved. The concept of *protected subsystems* in centralized systems is introduced and modified for use in the decentralized systems context. Protected subsystems serve as the model for discussing protection of external software. Some criteria for *acceptable* solutions are presented and some solution approaches are evaluated with respect to those criteria.

1.2.1 Protected Subsystems as a Paradigm for Externally Supplied Software

As noted in the preceding section, vendors require that proprietary software (programs and attendant databases) be protected from disclosure and re-distribution. In the extreme, disclosure may result in the complete exposure of the inner workings of the program, enabling the attacker not only to make copies of this software but also to understand the algorithms well enough to produce his own, equivalent software. Less severe disclosure may occur if only portions of the software are exposed or if only hints as to the algorithms employed in the program can be extracted, requiring significantly more effort by an attacker to generate equivalent software. On the other hand, it may be possible to re-distribute programs without knowing their content, e.g., if the programs were encrypted but the necessary cryptographic variables were not unique to a single client. For proprietary software that is rented or leased, a vendor may require a secure accounting capability, including a revocation mechanism, in support of usage- or

time-based billing policies. Finally, clients may wish to protect themselves from proprietary software, treating it as a potential *Trojan Horse.*

In the distributed systems context described above, users acting as vendors of external software have analogous security requirements. Here there may not always be a need to prevent disclosure of the programs (the algorithms used may not be considered proprietary) but databases associated with this software probably require concealment, as explained earlier. There is also a need to detect attacks that violate the integrity of the software, to prevent spurious information from being propagated throughout a distributed system application. For example, a query directed to a node maintaining a copy of a replicated database should either elicit a "correct" response or should go unacknowledged, rather than returning a response based on data that has been modified as a result of tampering. Although it might be suggested that externally supplied software should be protected from modification, it was noted above that merely detecting such attacks provides adequate security and is in keeping with the autonomous nature of the nodes. In particular, it is usually assumed that a user may "unplug" his node from the communication network, making all locally resident software and databases inaccessible to the remainder of the distributed system.

A general statement of security requirements for external software, from the standpoint of vendors, can be abstracted from the preceding discussion. The requirements are quite similar to those usually associated with *protected subsystems* in centralized systems, although some slight modifications are necessary to account for the scope of attacks to be considered. Schroeder [31] defines a protected subsystem as "a collection of programs and data bases that is encapsulated so that other executing programs can invoke only certain component programs within the protected subsystem, but are prevented from reading or writing component programs or data bases, and are prevented from disrupting the intended operation

of the component programs." From the standpoint of vendors, external software should be treated as protected subsystems with the caveat that modification (writing) and disruption by physical attacks need not be prevented, only detected. Note that detecting modification of code is often critical to preventing disclosure, e.g., if an attacker can undetectably modify code, he might effect disclosure by changing an address used in an output operation so that the program outputs itself!

The protected subsystem concept also models closely the security requirements of clients (users) with respect to external software. Restricting software so that it is granted appropriate access privileges to the minimal collection of data and programs required to perform its advertised function and so that it does not release that data to others is referred to as *confinement* [19]. Clients require confinement of externally supplied software to prevent release or modification of their own software and other externally supplied software. Clients also can employ confinement measures to restrict access of external software to various system resources. Thus interactions between external software provided by different vendors or between externally and locally supplied software should be characterized by *mutual suspicion* and protection from program-based attacks should be symmetric for both classes of software.

This discussion points out that vendors and clients have dual security requirements. Vendors require external software to be protected against program-based or physical attacks that result in release or undetected modification of information or invocation at other than specified external interfaces. They also require that this software not be re-distributable. Clients require external software to be confined, i.e., they require protection from program-based attacks launched by external software that would result in unauthorized release, modification or invocation of other externally supplied or locally produced software. Clients also require the ability to control the use of computer resources by external software.

Although these requirements can be combined into a fairly uniform statement about supporting mutually suspicious subsystems and confinement, the above-noted dichotomy between vendor and client requirements is important since it suggests an appropriate division of responsibility for achieving these requirements. The primary goal of this thesis is the design of computers that meet vendor security requirements, although systems that meet both sets of requirements are described in Chapter 5.

1.2.2 Solution Evaluation Criteria

In addition to meeting the security requirements noted above, protection mechanisms for use with externally supplied software in decentralized computers should meet some additional criteria.

Decentralization — The protection mechanisms must themselves be decentralized. The rationale here is that centralized approaches to providing protection tend to negate the advantages gained from decentralization.

Effectiveness — The mechanisms should provide a unified approach to meeting the security requirements over a broad spectrum of attacks. To provide a given level of security, based on an anticipated threat environment, only parameters of the mechanisms should be changed, not the mechanisms themselves.

Generality/Flexibility — The protection mechanisms should be applicable to a wide range of applications executing on a variety of system configurations and equipment. The mechanisms should not be dependent on a particular technology or equipment type.

Low Cost — The cost of equipment required to implement the protection mechanisms must not be prohibitive. The "bottom line" is that the use of the protection mechanisms should reduce losses by more than the cost of the mechanisms themselves.

Good Performance
: The addition of protection mechanisms to a computer often degrades performance. However, one must strive to minimize the severity of any performance degradation.

Transparency
: Protection mechanisms should be unobtrusive, so that writers of external software need not be very much aware of them. These mechanisms should have little or no effect on the design of external software.

This collection of criteria tends to rule out most measures currently employed to protect proprietary software. For example, use of service bureaus to offer external software is ruled out because it negates the advantages gained from decentralization. The *ad hoc* measures described in section 1.1.2 do not meet the effectiveness criterion. These measures also do not provide a unified approach to protection nor are they parameterizable to provide different levels of security for different environments. The protection measures described in the next section attempt to meet these criteria.

1.3 A Solution Approach

In order to meet the security requirements and evaluation criteria established in Section 1.2, a combination of physical, cryptographic and software protection measures are employed. Information stored or processed in computer system components is protected from physical attacks resulting in disclosure or undetected modification in one of two ways: by providing physical protection for a component or by using cryptographic techniques to conceal and error check information stored in or transmitted by the component. These basic techniques meet the security requirements of vendors of external software and are sufficient in situations where all of the external software executed on a computer is provided by a single vendor. In more elaborate systems, where external software is supplied by several vendors or

where external software interacts with client-supplied software, more conventional hardware and software security measures are employed in conjunction with the preceding techniques to provide the security required by mutually suspicious subsystems. This section briefly describes the proposed solution approach.

1.3.1 A System Model and Tamper-Resistant Modules

Before discussing the proposed solution approach, it is necessary to introduce a simple model of the computer systems of interest. The model, shown in Figure 1-1, consists of a processor (CPU), three levels of storage: primary memory (P-MEM), secondary memory (S-MEM) and transfer and archival storage (T&A), and various I/O peripherals, e.g., terminals or network interfaces. The only unusual component in this model is the transfer and archival (T&A) storage. This level of storage is used in two ways: vendors may *transfer* (distribute) copies of external software to clients using this level and external software may use it for secure *archival* storage, hence the name. (Vendors also may distribute external software via communication networks.) Storage media used at this level must be demountable and the files contained therein are usually viewed as outside of the file system proper. These two characteristics distinguish T&A storage from secondary memory, i.e., secondary memory need not be demountable and it contains the file system. The system components are connected by a bus used for addressing and data transfer, like the DEC UNIBUS [9] or the IEEE S-100 bus [11]. This architecture is typical of current personal and small business computers and serves as the model for the computer systems of interest.

If no precautions were taken, it is apparent that external software executing on this hardware could be attacked in a number of ways that would violate the security requirements of vendors. Physical attacks launched against the processor, bus or any of the storage devices could result in disclosure or undetected modification of

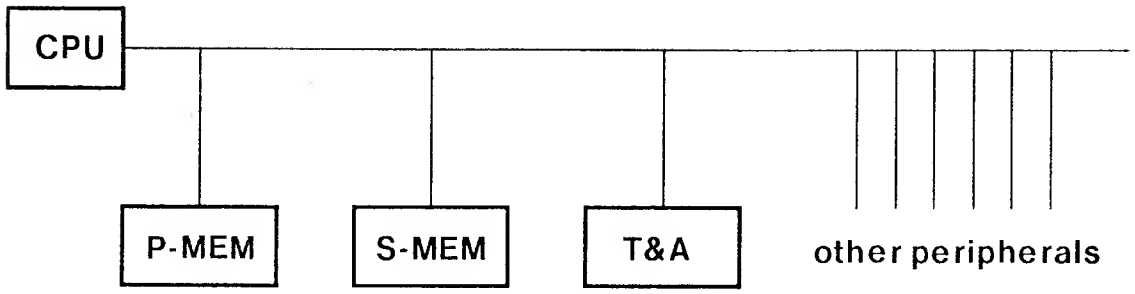


Figure 1-1: A Simple Model of the Systems of Interest

information. (Other peripheral devices included in the model are not security relevant since they do not store or process sensitive information.) It is obvious that some form of physical protection is required, at least for the processor if not other components. To evaluate the results of physically protecting portions of the system, the concept of a *tamper-resistant module* (TRM) is introduced. All information contained within a TRM is protected from disclosure and undetected modification in the following sense. As long as the TRM is intact, data inside the module cannot be discerned or modified by an attacker and if the TRM is breached the sensitive data within is destroyed (erased). The implementation of TRMs will vary considerably depending on the value of the external software being protected and the perceived sophistication of potential attackers. For example, packaging components on a single VLSI chip may provide adequate protection in some cases whereas permanently sealed, seamless metal containers may be required in other environments.

This thesis does not address the detailed problems of engineering tamper-resistant modules, but rather assumes that TRMs can be constructed to provide whatever level of physical security is required to protect external software in the

systems of interest. However, some observations can be made about characteristics of TRM-packaging. For example, TRM-packaging usually is not free and the cost increases with the volume of the TRM. Maintenance of components in a TRM may be difficult or impossible (if the TRM is permanently sealed). TRM-packaging may impose constraints on system growth and may limit equipment selection. Since sensitive data within a TRM must be destroyed if the TRM is opened, it may be difficult to package large quantities of non-volatile storage. Encapsulating demountable storage media in TRMs also may pose problems. These and other considerations suggest that packaging an entire computer within a single TRM, supplied by a vendor, is not an ideal way to protect external software provided by that vendor. Many of the shortcomings of TRM packaging can be avoided or at least mitigated by using TRM packaging in conjunction with cryptographic techniques.

1.3.2 Two Approaches to Protecting External Software

There are two basic ways to use cryptography in conjunction with TRM packaging: the *encrypted bus approach* and the *encrypted storage approach.* In the encrypted bus approach, the computer system is divided into several pieces, each contained in a TRM. Communication between the TRM-packaged pieces is provided by a physically unprotected bus. Here cryptographic techniques are used to secure inter-TRM communication over the unprotected bus. In the encrypted storage approach, the processor and some memory are packaged in a single TRM and all other storage is physically unprotected. Here cryptographic techniques are used to protect data held in physically unprotected storage and transmitted over the unprotected portions of the bus. Both approaches offer an effective, decentralized means of protecting external software but they differ in how well each meets other criteria.

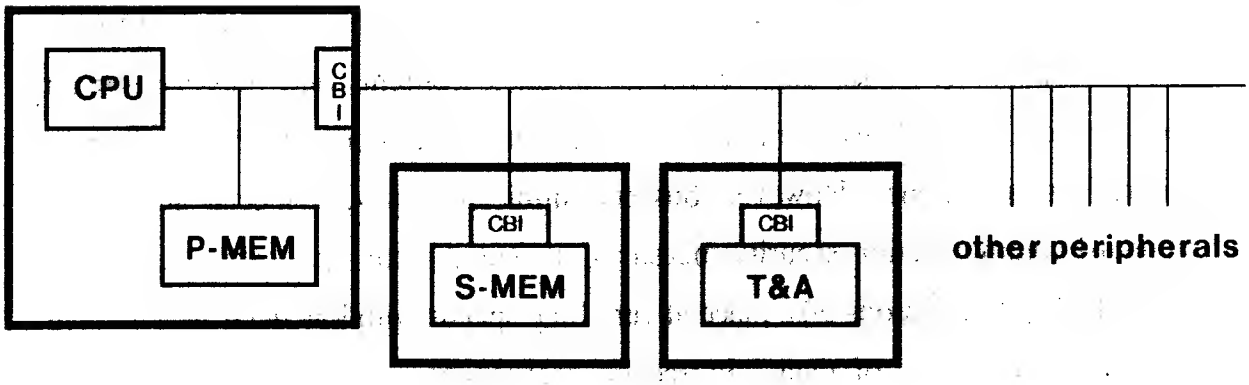


Figure 1-2: An Encrypted Bus Approach System Configuration

Figure 1-2 illustrates one of several system configurations based on the encrypted bus approach. In this configuration the processor and primary memory reside in one TRM whereas secondary and T&A storage devices are packaged in separate TRMs. (The bold boxes about these components represent the TRM packaging.) Communication among the TRMs is encrypted on the physically unprotected bus. Partitioning the system in this fashion reduces some of the TRM packaging problems, e.g., this design results in smaller TRMs and it supports expansion through adding or changing TRMs. It may even be possible to provide TRM-packaged demountable media in this design for T&A storage, although secure network communication offers a more practical means of distributing external software. Since all of the security relevant system components are protected by TRMs only the bus can be attacked. To counter these attacks, each TRM is equipped with a *cryptographic bus interface* (CBI). The CBIs employ cryptographic techniques to conceal and error-check data and addresses transmitted on the bus, thus preventing disclosure and detecting modification attacks.

In many respects the bus functions as a miniature communication network in which bus operations correspond to messages. The attacks to which bus operations may be subjected are the same as those encountered in general purpose communication networks, e.g., release of message contents and message stream modification [16]. Thus communication security techniques can be applied to secure bus operations. However, bus communication is very special and many standard communication security measures are not directly applicable here. For example, bus transactions take place at very high speeds with low delay and involve very small quantities of data. Protection mechanisms must be able to sustain maximum transaction rates, introduce little or no delay on transactions and minimize the number of additional bits transmitted for security purposes. Yet the data and addresses in bus operations must be concealed and checked to verify that they are properly ordered and not modified in transmission.

However, some of the special characteristics of bus communication simplify the task of securing bus operations. Most bus communication is very stylized in nature and this can be used to advantage in designing the encrypted bus protection measures. For example, one can take advantage of the fact that data transfers between primary memory and secondary or T&A storage involve *data aggregates* (e.g., disk sectors) that can be protected as a whole, rather than on a per-bus-operation basis. The high reliability and overall simplicity of bus communication simplifies bus protection measures, avoiding the need to provide efficient error recovery and/or to handle out-of-order message arrival. The cryptographic techniques developed for the encrypted bus approach are specially engineered to take advantage of the eccentricities of bus communication while keeping up with high transaction rates and minimizing overhead (delay and extra bits transmitted). These techniques also cope with the problems posed by having TRM-packaged and standard devices connected to the same bus.

Computer system designs based on the encrypted bus approach satisfy the criteria for decentralization, effectiveness, good performance and transparency and they are fairly general. Although this approach solves many of the problems encountered in trying to package an entire computer as a TRM, some problems still remain. For example, in partitioning the system, the pieces must not become too small or the cost of TRM-packaging and CBIs will become excessive. It probably is not practical to TRM-package demountable media, yet such media may be required for archival storage even if external software is distributed via networks. Problems in erasing large quantities of non-volatile storage and the need for periodic maintenance may preclude packaging some storage devices as TRMs. The need to enclose all security relevant components in TRMs also may limit equipment choices. Thus this approach is not as flexible as might be desired and the cost of TRM packaging may be a problem.

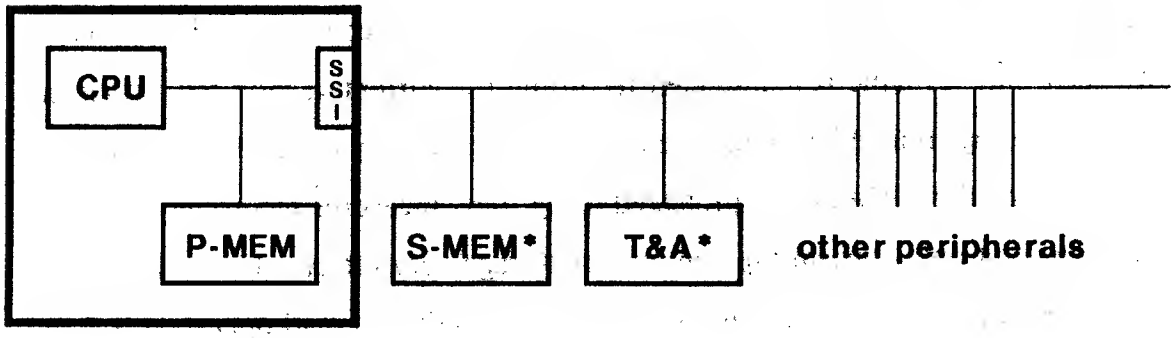


Figure 1-3: An Encrypted Storage Approach System Configuration

Figure 1-3 shows an encrypted storage approach system configuration comparable to the encrypted bus approach design in Figure 1-2. In this design the processor and primary memory are contained in a single TRM but secondary and

T&A storage devices and the bus connecting these devices to the TRM are all physically unprotected. (The asterisks in the figure indicate storage containing encrypted data.) The TRM is equipped with a *secure storage interface* (SSI) that employs cryptographic techniques to conceal and error-check data stored in these devices, to prevent disclosure and detect modification. This design provides excellent flexibility, generality and low cost. For example, the problem of building a TRM capable of erasing large quantities of non-volatile storage is avoided in the illustrated design since secondary and T&A storage is outside the TRM. All equipment outside the TRM is "off-the-shelf," allowing the clients great flexibility in selecting components and reducing costs. The fact that this design requires only one special device, an SSI, also contributes to its low cost and simplicity.

In the encrypted storage approach, data is aggregated into *storage units* that are read/written as an entity, e.g., groups of files that are archived and reloaded together (at the T&A storage level) or disk sectors (at the secondary storage level). Each storage unit is encrypted independently, in a fashion that is a function of both its address (or name) and a version tag, and an error detection code is associated with each unit. A table is maintained recording the current version tag associated with each storage unit. (This table is either contained wholly inside the TRM or it is stored outside the TRM and is protected using these measures recursively.) These techniques not only conceal the contents of storage very effectively, but allow the SSI to determine if a storage unit returned as the result of a read operation is from the correct location and if it is the most recent data stored at that location. The constraint that only the most recent copy of a storage unit be returned must be tempered in some circumstances for archival storage and it is not applicable to transfer storage (since such storage is read-only).

Except for designs in which primary memory is encrypted, i.e., located outside the TRM, the cryptographic techniques employed in the encrypted storage

approach do not encounter stringent performance constraints. The space required for error detection codes and for version tags is a very small fraction of that devoted to "real" data storage, except in the case of encrypted primary memory. If primary memory is encrypted, it is essential that the processor be equipped with a cache memory, to reduce the fraction of space devoted to overhead and to minimize the impact of delays imposed by encryption. Hierarchic structuring of the version tag tables for secondary storage and primary memory avoids the need to devote large amounts of space to VTTs and appropriate caching of portions of the hierarchy minimizes the performance impact of this structuring. Computer system designs based on the encrypted storage approach satisfy the criteria for decentralization, effectiveness, flexibility, low cost and are fairly general. These designs are not as transparent as those developed under the encrypted bus approach, largely due to the need to maintain VTTs. Their performance is generally good, except for those configurations in which primary memory is encrypted.

1.3.3 Two Approaches to Meeting Clients' Security Requirements

The preceding section briefly described two approaches to meeting the security requirements of vendors. These approaches protect external software supplied by a single vendor but they do not address the problems of meeting client security requirements or of executing external software from multiple vendors on a single computer system. These two problems are quite similar in that both require protection mechanisms that allow software from vendors and from the client to interact as mutually suspicious subsystems. This can be accomplished in two ways. A trusted third party can supply a TRM-packaged computer, based on one of the two approaches described in the preceding section, with a secure operating system. Both the client and the vendors must trust this computer to execute their software while meeting one another's security requirements. Vendors can transfer external

software to such computers either by forwarding it through the third-party or by using cryptographic techniques based on public-key ciphers [26]. This approach requires some standardization efforts so that external software from multiple vendors can be executed on third-party equipment under the secure operating system provided. The major problem here is that both vendors and clients must rely on the third-party to produce a secure operating system and a secure TRM-based computer.

An alternative to this approach is to allow each vendor to supply his own TRM-packaged processor and memory and to connect these modules together under the control of a client processor. Figure 1-4 illustrates one way this could be accomplished. In this example two vendors have supplied TRMs, each containing a processor and primary memory. Secondary and T&A storage are shared among the TRMs and the client processor. The client processor controls access to these and other shared system resources through an *access control bus coupler* (ACBC). The access control mechanisms used here are similar to those employed in centralized systems but are somewhat simpler to implement here due to the hardware isolation provided by the design. This approach has the advantage that no mutual trust is required since each vendor supplies his own TRM. This approach allows vendors to select their own processor base but some standardization of TRM interfaces and operating system interfaces is still required. It also remains to be seen if the cost of TRMs can be reduced to a point at which this becomes economically feasible.

In distributed systems members of the user community need to act both as clients and as vendors in writing and using external software. In fact, a user may act as both client and vendor for the *same* software. A combination of the preceding two approaches can be employed to meet this complex security requirement. Each node in the distributed system can consist of a client processor and a TRM supplied by a third-party, configured as in Figure 1-4. The third-party TRM is used to execute

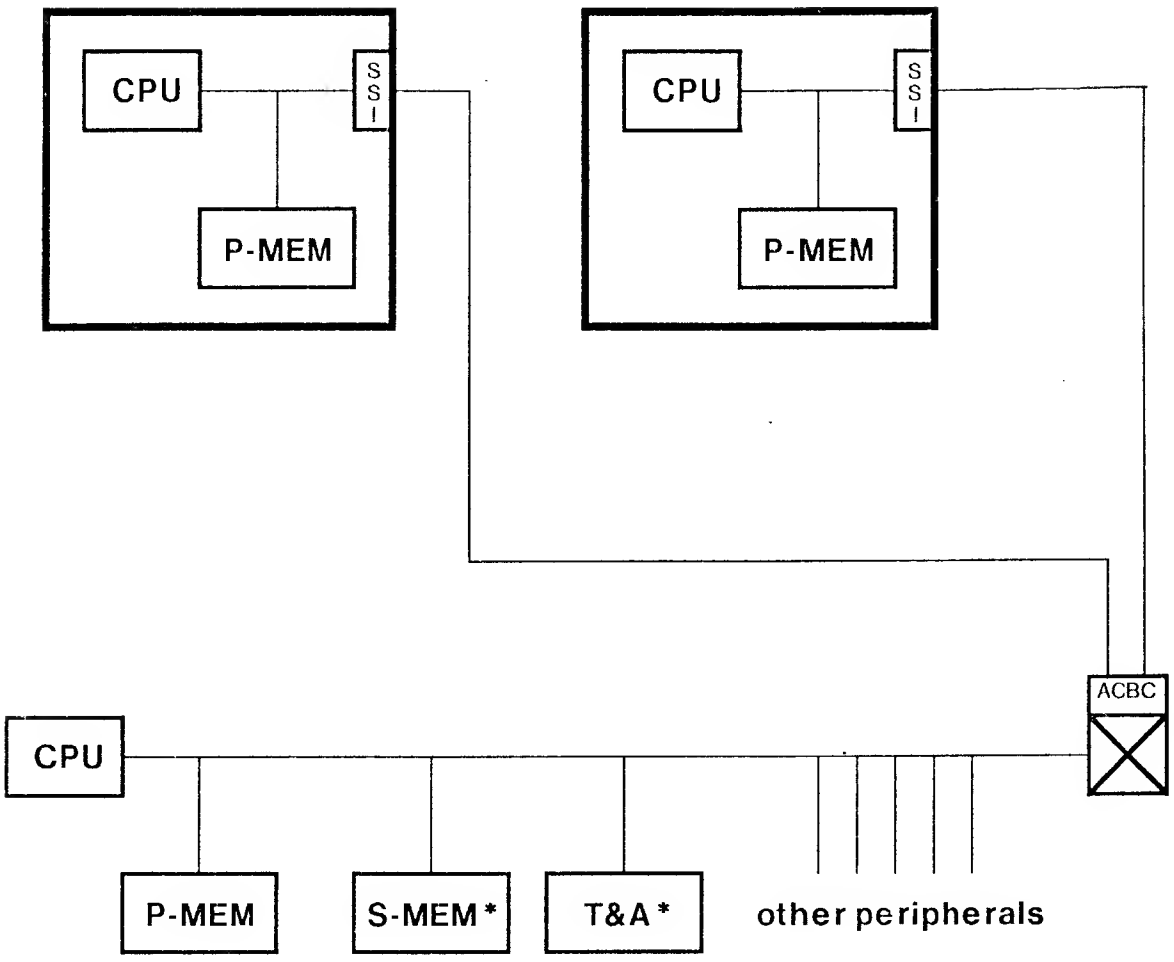


Figure 1-4: A Multi-TRM System Configuration

external software supplied by other members of the user community, treating each user as a separate vendor. To solve the problem of vendors being their own clients, another third-party TRM is used to distribute the locally produced external software. In this fashion a would-be vendor submits his software (source code) to an *installation server* TRM which compiles code and distributes it securely to the TRMs at the user nodes. Since this software is not proprietary, the client-users can

be allowed to review the source code and decide if they want to use the software. In this fashion users can decide for themselves if some distributed application implements an advertised security policy that achieves their requirements for confinement.

1.4 Related Work

The central topic of this thesis, the development of protection measures for use with externally supplied software in decentralized computing facilities, has received little attention in the open literature. The general problem of protecting information stored in centralized computer systems has been the subject of much research. (See [29] for an excellent bibliography.) Most of this research deals with protection of information from program-based attack or with controlling physical access to central computer facilities. Although the concepts developed in such research are applicable to the problem of protecting external software in decentralized systems, most of the detailed mechanisms developed for centralized systems are not relevant to this "physically hostile" environment. The major exception is the use of a secure operating system to provide protected subsystems in third-party, multi-vendor computer system designs. Multi-vendor systems in which each vendor supplies his own TRM also may make use of some conventional access control mechanisms in managing shared resources.

There has been relatively little published research dealing with protection problems in distributed systems. Much of this research assumes that the nodes that make up the system are under the control of a single authority, e.g., see [5], as opposed to the autonomous nodes considered in this thesis. In designing distributed systems composed of autonomous nodes, usually the tacit assumption is made that software executing at remote sites cannot be protected from physical or program-

based attack by the user at the node if the concept of nodal autonomy is to be supported. Thus the protection measures developed for such systems tend to be limited in scope [33]. One report [20] proposed using cryptographic methods to protect data objects in distributed systems, allowing the objects to be transmitted to nodes for examination while being able to detect modification of the objects upon return to their "owner." However this is a very limited facility that does not address the full range of protection problems described and solved in this thesis.

A substantial body of literature deals with legal protection for proprietary software (see [21]), but not with the development of technological measures to protect such software. A notable exception is a patent [1], issued in September 1979, which proposes cryptographic mechanisms for protecting proprietary software for use with personal computers. The patent describes a microprocessor designed to execute enciphered programs. This design is superficially similar to the encrypted storage approach configuration illustrated in Figure 1-3 but it differs in a number of ways. For example, the protection provided by this patented design applies only to object code and read-only databases, not to modifiable databases. (The inventor claims that the mechanisms could be used to protect such databases but significant cryptographic weaknesses would become apparent in such applications.) This restriction precludes a number of applications both for proprietary software and for distributed systems software.

The same cryptographic limitations that preclude use of this design for modifiable databases also restricts the design to executing only one program per microprocessor chip. This is in marked contrast to the system designs proposed in this thesis each of which is capable of executing an essentially unlimited number of program products from vendors. In fact, the cryptographic techniques presented in the patent are capable of concealing no more than one primary memory image worth of code/data, so secondary and T&A storage mechanisms are inapplicable

here. More importantly, this patented microprocessor design includes no facilities for detecting modification of code or data. As noted earlier, the lack of such measures permits some attacks that could result in disclosure of the code or data, so this design does not even provide complete protection against disclosure. The lack of modification detection mechanisms also severely limits the range of applications which can be protected by this design, e.g., the design is incapable of providing secure accounting or revocation facilities or of supporting distributed systems software as described above. Thus this patented design differs in many respects from those presented in this thesis.

The areas which are most directly related to this thesis are cryptography and communication security research. This thesis does not develop cryptographic algorithms but it does rely on an understanding of basic cryptographic techniques and of characteristics of modern ciphers, e.g., the Data Encryption Standard [23] and the RSA public-key algorithm [26], in developing the encrypted bus and encrypted storage approach of protection mechanisms. The problems of protecting information transmitted on a bus in the computer systems of interest differ somewhat from those encountered in protecting information in general purpose communication networks, but communication security research does offer some help. For example, research in this area provides a taxonomy of threats that are applicable to the thesis problem and offers techniques for dealing with these threats in general purpose communication environments. Some of these techniques are directly applicable to the problems encountered in this thesis and others can be modified to meet the specialized requirements encountered in this context.

Some research has been carried out on the use of cryptography to protect files in centralized systems. Commercially available software developed at IBM [12] provides key management facilities and encryption/decryption primitives that can be used with files on secondary storage, but these mechanisms must be explicitly

invoked by the user and no higher-level, encryption-based protection mechanisms are provided, i.e., there is no specific support for mechanisms to detect modification of data. Moreover, the elaborate key management facilities provided by this software is designed for multi-user centralized systems, not the single-user, decentralized systems which are the topic of this thesis. Thus this work has very little relationship to the topic of this thesis. Other researchers [18, 27] have suggested using cryptographic techniques to protect information stored (and executed) at centralized systems, but these suggestions have not been accompanied by detailed proposals or even thorough analyses of the security requirements. It is easy to postulate encryption as a means of protecting information in this context but, as this thesis illustrates, there are a number of difficult problems that must be solved in implementing such mechanisms.

In summary, the problem of designing protection mechanisms for use with externally supplied software in decentralized computing environments has received little attention. The only work that parallels this thesis is that of a patented microprocessor design which, as noted above, does not address the full range of problems described and solved in this thesis. Research in protection of information in centralized systems, communication security, cryptographic file security and distributed system protection mechanisms all contribute in some fashion to the work described in this thesis but this work studies and solves problems that have not been addressed previously.

1.5 Thesis Outline

Chapter 2 explores in detail the system model introduced in this chapter. The chapter projects values of various parameters for processors, busses and storage and peripheral devices that might be used in the systems of interest over the next 3-5

years. This chapter also examines the concept of tamper-resistant modules in greater depth, noting some of the problems that may arise in engineering such modules. The simplest approach to protecting external software based on the use of a TRM is described and evaluated. The chapter concludes with a brief discussion of cryptography and a simple application example, secure network-based distribution of external software. The protection mechanisms developed in Chapters 3 and 4 employ cryptographic techniques, so this discussion is intended as background for the reader who may be unfamiliar with fundamental cryptographic techniques.

Chapter 3 develops designs for protecting external software based on an encrypted bus approach. It contrasts security requirements for this approach to those usually associated with communication systems. The chapter develops cryptographic-based protection mechanisms to secure transactions on a physically unprotected bus connection TRM-packaged devices that form a computer system. In developing these mechanisms, special attention is paid to minimizing the impact of protection measures on the performance and overall cost of the computer system. System initialization procedures, error response and recovery measures and procedures for adding new TRMs to a system are presented. This chapter describes ways of interfacing non-secure devices to these encrypted bus systems .

Chapter 4 develops system designs based on an encrypted storage approach. The security requirements in this approach differ somewhat from those in the encrypted bus design. These differences are examined through the use of an abstract model that captures the essential features of this approach independent of the system configuration employed. Cryptographic-based protection mechanisms are developed to secure data held in physically unprotected storage. The protection mechanisms employed here differ noticeably from those developed in Chapter 3. Again, special attention is paid to minimizing the impact of these protection mechanisms on system performance and cost.

Chapter 5 explores the problems of developing computer systems that execute software supplied by multiple vendors and of meeting user security requirements in the context of systems executing external software. This chapter uses the system designs of chapters 3 and 4 to achieve these dual requirements. These requirements can be met in two ways, either through the use of third-party supplied TRMs with trusted operating systems or through the use of separate TRMs (one per vendor) combined into a single computer system. Both of these approaches are described and evaluated in terms of cost, effectiveness and acceptance by users and vendors.

Chapter 6 summarizes the results of the thesis, examines the applicability and limitations of the proposed mechanisms and suggests possible directions for further research in this area.

1.6 How to Read This Thesis

Theses can be read at a number of levels, ranging from cursory perusal to critical, in-depth analysis. Those who wish only an overview of the research described in this thesis probably should read only this introductory chapter and the concluding chapter. Such readers are already more than half-way through if they have not cheated (by skipping material before this section). Brave souls who desire a detailed understanding of all the protection mechanisms developed in the thesis will have to wade through each chapter, section and subsection. However, individuals with some understanding of cryptography may skim the discussion of this topic presented in section 2.3. Special provisions have been made for readers seeking a thorough understanding of this research but not wanting to examine all of the proposed mechanisms in detail. At one or more points in Chapters 3, 4 and 5, instructions have been included to direct the reader around detailed discussions of specific protection mechanisms. One can gain a fairly good understanding of this research

by following these directions, even if all of the detailed discussions are avoided. As a further aid to the reader, a list of acronyms used in this thesis is provided as an appendix (page 248).

Chapter Two

The System Model, TRMs and Cryptography

This chapter begins by describing in greater detail the computer system model introduced in section 1.3.1. Variations on the basic model are introduced and projected characteristics of devices in these systems are extrapolated from current device specifications. This model provides an engineering context for the design and evaluation of the protection mechanisms explored in the thesis. Next, the chapter explores the use of tamper-resistant modules (TRMs) to physically protect security-relevant system components and thus protect external software, meeting the requirements of vendors. A simple system design employing a single TRM can meet vendor security requirements, but there are a number of limitations associated with this simple design. To overcome these limitations, more elaborate designs combining TRMs and cryptographic techniques are developed in Chapters 3 and 4. This chapter concludes by introducing the reader to some cryptographic concepts and examining cryptographic techniques for use in the latter chapters.

2.1 The System Model Revisited

A simple model for the computer systems of interest was introduced in Section 1.3.1. This model, reproduced in Figure 2-1, and variations on it are described in greater detail in this section. The model provides a framework in which detailed designs of protection mechanisms are developed and evaluated and it includes only those details that affect these mechanisms. For example, most details of bus arbitration are ignored as they are largely irrelevant to the proposed protection mechanisms, whereas timing characteristics of devices in the system are presented

since they are necessary in evaluating the performance impact of such mechanisms. This model attempts to embody the high level architecture of personal and small business computers that will be constructed in the next 3-5 years. However, differences between this model and computers actually produced need not preclude the adoption of the protection mechanisms developed in the thesis. In fact, the protection mechanism designs that are most likely to prove feasible are largely independent of details of processor and primary memory operation. Thus, although the system model attempts to capture salient features of real computers, deviations from this model do not affect all the protection mechanisms proposed in this thesis.

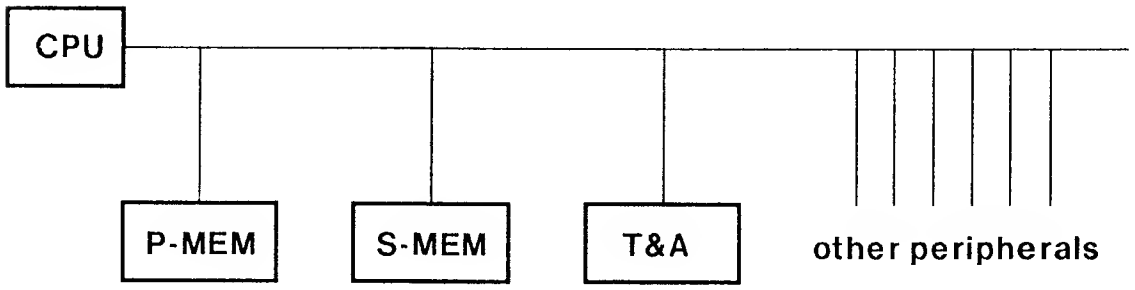


Figure 2-1: The Basic Model for the Computer Systems of Interest

Before proceeding to a discussion of variations on this basic model, some additional comments are in order. In Figure 2-1 and other system configuration diagrams each storage system component is depicted as a single box. This is not meant to imply that in every case there is but one of each of these devices nor that multiple instances of a device are packaged together. In the basic system model there is only one processor (CPU) but there may be multiple, independently packaged instances of the storage devices. In particular, when storage devices containing sensitive data are TRM-packaged, additional, non-TRM-packaged

devices may be used to hold client data since vendors are not trying to protect this data from physical attack. This device replication is not required for vendor security but may be preferred by clients since it gives them full access to their data. (This dual packaging strategy is not applicable to transfer storage since it is used exclusively by vendors.) Thus, these configuration diagrams illustrate minimal implementations.

In section 1.3.1 there was a brief discussion of how secondary storage (S-MEM) differs from transfer and archival storage (T&A). It was noted that transfer and archival storage is always demountable whereas secondary storage may be non-demountable. Thus these two types of storage are not necessarily distinguishable based on the devices used to implement them, i.e., a demountable disk might serve as either transfer and archival or secondary storage. A second distinguishing feature is that files on T&A storage are viewed as being *outside* of the file system maintained on secondary storage. The assumption here is that program files are transferred into primary memory for execution from the file system (via swapping or demand paging). Portions of data files are read and written by transfers between primary memory and secondary storage, e.g., disk sectors may be the object of such transfers. Externally supplied software distributed to a client on transfer storage media is moved to a permanent home on secondary storage before use. A sensitive file on secondary storage may be recorded on secure archival storage media and later can be *reloaded*, i.e., copied to the file system under its original name.

There are three possible reloading constraints associated with files maintained on secure secondary storage: *unconstrained*, *non-reloadable* and *most recent only*. Some files have no constraints on reloading, i.e., the client is free to reload any archived copy of the file. An object code file produced by a proprietary compiler might fall into this class since the vendor has no concern over which version of the file is executed by the client. Other files are non-reloadable, i.e., under no circumstances

should these files be archived and later reloaded. Accounting files used by proprietary software may fall into this category since if they were reloaded the client could "turn back the clock" on the billing function they provide. Special precautions must be taken to ensure the reliability of these files and these precautions may significantly increase the space occupied by the file. Vendors also may require some files to be archived and reloaded together by the operating system (to enforce some consistency constraints) and these can be grouped into *archival units* on archival storage. The same concept can be applied to files that make up external software packages, yielding *transfer units* on transfer storage. Ways in which these groupings can be implemented securely are examined later.

In between these two extremes are files that may be reloaded only from the most recent archived copy of the files. For example, a database may be periodically checkpointed (archived) and a small transaction log may keep track of the updates that take place between checkpoints. The database should be reloaded only from the most recent archived copy and the small transaction log can be non-reloadable. These reloading constraints apply not only to individual files but also to groups of files that must be archived and reloaded together, to ensure consistency across file boundaries. (Such consistency also may be achieved explicitly by including some information in each file that binds it to the other files archived at the same time.) Even if there are no constraints with respect to timeliness associated with reloading a file (unconstrained), it may be required that other files archived at the same time must be reloaded along with this file. Thus even *unconstrained* files may have some constraints on reloading.

2.1.1 Variations on the Basic Model

The computer system pictured in Figure 2-1 employs a single, general purpose bus to interconnect all of the system devices. Figure 2-2 illustrates a variation on

this model, a *dual-bus system* in which primary memory is attached via a dedicated *memory bus* whereas other devices are attached to an *I/O bus* and the two busses are connected via a *bus coupler* at the processor. (The bus coupler provides functions necessary to mate the two busses, e.g., buffering and inter-bus arbitration.) A dual bus system offers several advantages over a single bus system. The memory bus, since it is quite short and since it is specialized in function, can be made faster than a general purpose or I/O bus, thus reducing effective access time to primary memory. The I/O bus is used to interconnect devices with less stringent performance requirements and thus can be slower than a general purpose bus. In this way more expensive, high speed bus interfaces are employed only on the memory bus (2 interfaces) and less expensive bus interfaces are used on the I/O bus where many more interfaces are required. This configuration also reduces contention on both busses, further improving performance.

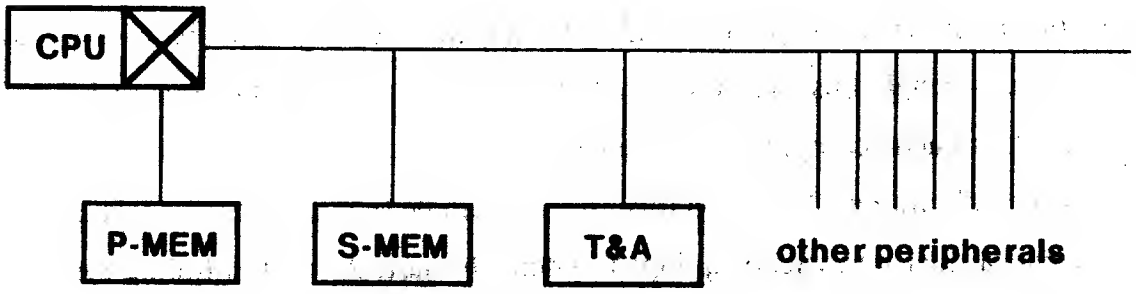


Figure 2-2: A Dual Bus System Model

Dual bus systems provide improved performance at the cost of a bus coupler and two high speed bus interfaces. This performance gain entails some cost and since high performance is not a major design parameter for the systems of interest, one expects to see both single and dual bus systems in practice. Another way to improve

system performance is to add a cache memory to the processor. (System model diagrams do not explicitly illustrate the presence of a cache at the processor.) The major motivation for using cache memory is that it reduces the effective access time of primary memory. As processors in the systems of interest become faster, inclusion of cache memory will probably become appropriate. Moreover, use of cache memory allows somewhat slower, cheaper primary memory to be employed with only a minimal effect on effective access time. This is an important feature as processor costs will be small relative to primary memory costs in many of these computer systems. Finally, use of cache memory reduces bus contention and may eliminate the need for a very high speed bus, i.e., one capable of keeping up with processor-generated references to primary memory.

Again, the performance gain achieved here is not without cost. The addition of cache memory to a processor is a non-trivial engineering task and the cost of the resulting system is correspondingly increased. Thus one expects to encounter both cache-equipped and cacheless systems in practice. A cache can be added to a processor in either a single or dual bus system, yielding four basic system configurations: single bus cacheless, single bus cache-equipped, dual bus cacheless and dual bus cache-equipped. In general, system performance improves with successive configuration choices on this list, i.e., a single bus, cacheless system is the slowest and a dual bus, cache-equipped system the fastest. In illustrating system configurations, if the choice between single and dual bus designs or the inclusion or omission of a cache is irrelevant to proposed security mechanisms, the generic model of Figure 2-1 will be used. Otherwise, specific bus configurations will be shown and the inclusion or omission of a cache will be noted in the text.

2.1.2 Processor and Storage System Parameters

Most details of processor operation are irrelevant to the model but a few parameters are critical to the formulation and evaluation of design options. One of the most important parameters is the processor *word* size, i.e., the number of bits of data normally fetched and transformed by the processor. A word size of 32 bits is projected for the systems of interest. This is a larger word size than most personal computers employ at this time, but already there are single chip processors with 32-bit registers, e.g., the MC68000 [22], and full 32-bit microprocessors will probably be announced before the end of 1980. The processor should be capable of directly addressing about 16M-32M bytes of primary memory, to take advantage of the continuing improvements in memory technology. Bus addresses should be a little less than 32 bits, to support byte addressing of primary memory (24 to 25 bits) and for control of peripheral devices. The size of these addresses and the word size suggests that one set of bus lines should be used alternately for addresses and data, to reduce the cost of bus interfaces. This is especially important for the general purpose and I/O busses since a number of devices will be connected to these busses.

If the processor is equipped with a cache memory, several additional parameters come into play: cache size, line width and update scheme.[3] A survey of existing 32-bit, cache-equipped processors turns up cache sizes ranging from 8-32 Kbytes and line width of 8-32 bytes. As noted earlier, the systems of interest are not intended for extremely high throughput, so the projected cache size for these systems is 8 Kbytes. For most systems a line width of 8 or 16 bytes (2 or 4 words) will be appropriate but a 32-byte line width will be required in support of some encrypted storage protection mechanisms. Since the systems of interest generally support only a single user, the hit rate for a cache of this size may be in the range of 95-98% [6].

---

[3] A *cache line* is the group of words treated as a unit for addressing and replacement purposes. Within the cache, there are a number of *cache line frames*, each capable of holding one line.

Cache memory control logic will employ one of two schemes for updating the contents of primary memory: *write-through* or *write-back*. In a write-through cache, a *write* to a word in the cache is propagated to primary memory immediately, so that primary memory and the cache remain "in sync." (In fact, the update of primary memory normally is buffered by the cache so that the processor does not have to wait for the primary memory access to complete, so there is a short time window when the two are not in sync.) If the target of a *write* is not in the cache, then the update takes place only in primary memory, i.e., the cache is not affected. In a write-back cache, *writes* are effected only in the cache, i.e., an attempt to modify a word not in the cache results in a fetch of the appropriate cache line from primary memory. Updates are propagated to primary memory only when modified cache lines are evicted as part of the cache replacement strategy. (Note that an entire modified cache line is copied into primary memory; there is no attempt to keep track of which words in the line were modified.) In a write-back cache anywhere from 20-60% of the misses result in eviction of modified lines, i.e., the evicted line is written into primary memory. Unless otherwise stated, caches in this thesis are assumed to be write-through.

To estimate the performance characteristics of the processor and various levels of storage, one must adopt some *rules of thumb*. Recent trends in semi-conductor technology provide several such rules for projecting the performance and cost of the systems of interest [2]. These projections are useful in that they provide a basis for evaluating proposed designs in terms of technological (and economic) feasibility. For example, one rule of thumb notes that the component count per IC chip approximately doubles every year and memory chip capacity quadruples every two to three years. At the same time, raw speed of IC chips doubles every five years. As production techniques are refined the cost of producing chips with constant performance characteristics drops by about 20% per year. Using these rules of thumb, one can extrapolate from current product specifications to project some of the characteristics of systems that will come into existence over the next 3-5 years.

Based on these trends, the minimum instruction execution time for processors in the systems of interest should range from about 100ns (10 MIPS maximum) for a high performance multi-chip CPU (the "top of the line" in this class of systems) to about 600ns (1.6 MIPS maximum) for a slow, single chip processor (a "low end" entry in this class). It is assumed that the fastest instructions are register-to-register operations, no memory references are involved, so this time is also taken as the minimum time between processor-generated memory references. The mean time between processor-generated memory references is assumed to be about a factor of 3 or 4 greater than this minimum, accounting for longer instruction execution times and references for instruction operands. This yields processors with average speeds ranging from 0.4 to 3.3 MIPS (assuming matched primary memory access times as described below). For the storage components of the system, there are a number of relevant device characteristics: access time and transfer rate, mean time between references, storage capacity of the device, size of data aggregates transferred to and from the device and the mean time between failure (MTBF) of the device. In general, going from the lowest level in the storage hierarchy (cache memory) to the highest (T&A storage) the access time, mean time between references, capacity and data aggregate size all increase whereas the MTBF and transfer rate decrease.

The volatility and demountability of storage devices are also relevant to the system model. Cache and primary memory are constructed from solid state components and are volatile whereas secondary memory and T&A storage are non-volatile. Only T&A storage is required to be demountable but secondary storage may also be demountable, depending on the technology employed. Note that even though magnetic bubble memories may see increased application in this time frame, such memories are not expected to be price competitive with removable magnetic media for many applications and thus will not significantly displace such media. In fact, the recent improvements in non-demountable disks, e.g., *Winchester* technology disk drives, make it likely that magnetic bubble memories will not

significantly displace disks for some time. Thus the predominant form of secondary storage employed in these systems is likely to be magnetic disks. Also, not all system configurations will provide separate devices for secondary and T&A storage, thus demountable media may serve a dual role in some systems.

Now consider projected values of some these parameters for devices at various levels in the storage hierarchy. In high performance systems employing a cache, the effective access time will be about the same as the minimum instruction execution time. (The memory chips used in caches are static RAMs so the cycle time and access time are the same.) This access time includes checking to see if the requested word is in the cache and the transport delay between the cache and processor. Thus a processor with some instruction lookahead facilities can maintain a continuous stream of references to the cache for minimum time instructions. This suggests an effective cache access/cycle time of about 100ns, which yields a transfer rate of 320 Mbits/s. Access time for primary memory (using 64-256 Kbit chips) should range from about 100ns to 200ns, exclusive of bus transport time, with cycle time about twice access time. Bus time will add some 200ns to 300ns to this access time (for transport), yielding an effective primary memory access time of about 300-600ns, so the maximum primary memory transfer rate ranges from about 106-213 Mbits/s. (This transfer rate assumes a non-interleaved memory; cache-equipped systems will require at least two-way interleaving for quick transfer of cache lines, increasing the transfer rate.)

In a cache-equipped system, the effective memory access time seen by the processor is determined by the access times of the cache and primary memory, by bus transport time and by the hit rate. A cache-equipped system using fast (100ns access time) primary memory and a fast (100ns transport time) bus can achieve an effective average access time of 104-110ns, based on a 95-98% hit rate. For a cache-equipped system using slower primary memory (200ns access time) and a slower bus

(200ns transport time), the effective average access time is 110-125ns, based on this hit rate range.[4] This illustrates the enormous improvements that can be obtained by inclusion of a cache memory. Even if performance is not a critical concern, economics may dictate use of a cache since it allows use of slower, cheaper memory chips for primary memory. At this time, the location of the "break even" point, based on the cost of equipping a processor with a cache versus the cost of memory chips and the anticipated size of primary memory, is not obvious.

For secondary storage the access times and transfer rates vary considerably depending on the technology employed. For example, magnetic bubble memories may provide average access times of 10-15ms and transfer rates of 0.1-1.5 Mbits/s whereas fixed disks may exhibit average access times of about 70ms and transfer rates of 10-15 Mbits/s. Bubble memories, using 4-16 Mbit chips, may be configured as small capacity storage devices (4-16 Mbytes) whereas hard disks may contain up to 100 Mbytes. Devices used for T&A storage tend to be relatively slow, at the low end of the range for secondary storage devices. For example, floppy disks may exhibit access times on the order of 100-400ms and transfer rates of 0.5-1.0 Mbits/s. Capacity for floppies may grow to 5-10 Mbytes using double sided, double density recording technology. For all of these secondary and T&A storage devices the (usable) record size is expected to be about 512 bytes. These characteristics of the computer systems of interest are collected in Table 2-1.

---

[4]This effective average access time calculation assumes that on a *cache miss* the first word fetched is the one which caused the miss and that subsequent references to words in the fetched line occur at cache speed. This second assumption may not hold for long cache lines (>4 words) or if a slow bus and slow primary memory are used.

System Characteristics

- Processor and Bus

  *word length: 32 bits

  *minimum instruction time: 100-600ns (1.6-10 MIPS)

  *average instruction time: 300-1800ns (.4-3.3 MIPS)

  *bus cycle time: 100-200ns

  *multiplexed data/address bus lines: 32

- Cache (optional)

  *access/cycle time: 100ns

  *line width: 8, 16 or 32 bytes

  *capacity: 8 or 16 Kbytes

- Primary Memory

  *access time: 100-200ns

  *cycle time: 200-400ns

  *capacity: 64K-16M words

- Secondary and T&A Storage

  *access time: 10-400ms

  *transfer rate: .1-10 Mbits/s

  *capacity: 5-300 Mbytes

  *record size: 512 bytes

Table 2-1: Characteristics of the Computer Systems of Interest

2.1.3 Other Peripherals

In Figure 2-1 peripherals other than storage devices are lumped together at the end of the bus under the heading "other peripherals." This heading includes terminals, bulk I/O devices and communication facilities, e.g., network interfaces. These devices are not described in detail since their operation is not critical to the security of external software. For example, external software that interacts with a user via a terminal must be prepared to accept any input from the user and thus no tampering with the terminal should affect the secure operation of the software. The same argument holds for hardcopy output devices and even for network interfaces. (If external software requires secure communication facilities, these facilities will be provided within the TRM containing the processor.) In designing mechanisms to protect external software, provisions must be made to accommodate I/O devices, i.e., these devices must still function properly in conjunction with protection mechanisms.

Only two I/O devices exhibit high enough transfer rates to warrant further discussion: network interfaces and bit-map displays. For most personal and small business computers the network interface will be telephone based and thus is restricted to relatively low bandwidth, e.g., less that 10 Kbits/s. However, in distributed systems, high speed local area networks will probably be employed and the bandwidth could be in the neighborhood of 10-20 Mbits/s. This transfer rate is equal or greater than that of many secondary storage devices and thus constitutes a significant contribution to bus utilization. Many systems may be equipped with bit-map displays in the future. These displays associate with every pixel on the screen one bit in a display memory, typically on the order of 128 Kbytes. (Color bit-map displays associate several bits with each pixel.) The data transfers required to manipulate the display may be limited primarily by memory access time, so these displays are capable of very high transfer rates and they can become dominant users of a general purpose or I/O bus.

2.1.4 Basic Bus Characteristics

The busses (general purpose, I/O and memory) employed in the model are abstracted from conventional designs such as the DEC UNIBUS and the IEEE S-100 bus. Only those characteristics of bus operation that directly affect the design of protection mechanisms are included in the model. The bus consists of a collection of bidirectional lines for transmitting addresses, data and control information, as detailed in Table 2-2. (Additional lines are provided for timing, arbitration, power, etc. but are not included the model.) The general purpose and I/O bus are asynchronous or pseudo-synchronous whereas the memory bus is assumed to be synchronous. A *bus cycle* is the time interval required to perform a *bus operation.* There are four bus operations: **PRESENT-ADDRESS**, **PRESENT-DATA**, **ACKNOWLEDGE** and **ERROR**. The first is used to place an address on the bus, the second does the same for data (or an interrupt vector) and the third acknowledges receipt of data. The last operation, **ERROR**, is described below.

Bus cycles are well defined for synchronous and pseudo-synchronous busses; for asynchronous busses the minimum time required for a bus operation as described above will be referred to as the bus cycle time. For the systems of interest the bus cycle time will range from about 100ns for a memory bus to about 200ns for general purpose or I/O busses. An arbitration mechanism, which may proceed in parallel with data transfers, is used to select the next device to use the bus, i.e., the *bus master.* (Although arbitration is an important aspect of bus design, all of the commonly used bus arbitration schemes are essentially equivalent from the standpoint of security and thus no specific arbitration scheme is included in the model.) Once granted the bus, the bus master uses two or more operations to complete a *bus transaction*, e.g., a data transfer, with another device, the *slave.* (In asynchronous and pseudo-synchronous busses a handshaking protocol usually is employed to allow both slave and master to control the duration of the transaction.)

BUS LINE	DESCRIPTION
A/D0-31	used to transmit addresses and data
PARITY0-3	used to parity check lines **A/D0-31**
ADDR	asserted when an address is on lines **A/D0-31**
DATA	asserted when data is on lines **A/D0-31**
INT	asserted when interrupt vector is on lines **A/D0-31**
READ	asserted during **read** transactions
WRITE	asserted during **write** transactions
EXT	asserted during *extended* transactions
ACK	asserted by a slave to acknowledge a **write** or **interrupt**
ERROR	asserted by a slave to indicate a bus operation error
RESET	asserted to reset the device selected by lines **A/D0-31**

Table 2-2: Bus Lines for the System Models

The **ERROR** operation noted earlier is issued by a slave if a transaction cannot be successfully completed, even though the master uses a timeout to detect the failure of a slave to respond.

Associated with each device on the bus are one or more addressable cells from or to which data is read or written (or both). A device examines addresses placed on the bus to determine if one of its cells is the target of an operation. In the case of primary memory these addresses correspond to storage cells whereas for other devices they represent control and status registers. The processor writes into a control register to initiate an operation and reads from a status register to determine the outcome of the operation. For example, the processor initiates a direct memory access (DMA) transfer of data from a disk to primary memory by writing the (disk) source address, the (primary memory) target address and the number of words to transfer into appropriate disk control registers. The disk then transfers data to primary memory, one word at a time, indicating completion of the transfer by setting an appropriate value in its status register and by generating an interrupt. Devices that transfer very small quantities of data, e.g., character-at-a-time I/O devices, often use device registers to hold the data rather than employing the DMA technique described above. In such cases the device generates an interrupt and the processor transfers data between primary memory and the device register.

In systems employing a dedicated memory bus, this bus is assumed to be quite similar to the general purpose and I/O busses described above. There will be no arbitration mechanism because there is only one bus master, the bus coupler (processor), and there is no need for interrupts. The memory bus will be synchronous with transfers taking a known period of time, since the memory provides a uniform access time. Thus a memory bus is somewhat simplier than a general purpose I/O bus. The functions provided by a bus coupler used to interface these two busses will vary depending on the system design. For example, the coupler may provide some buffering for speed matching, to account for differences in the number of bus cycles required for operations on the two busses and to manage arbitration across the two busses. On a store into primary memory by a device on the I/O bus, the bus coupler can generate an **ACKNOWLEDGE**

immediately and carry out the transaction on the memory bus asynchronously. On primary memory fetches initiated by devices on the I/O bus, the bus coupler can prefetch data in anticipation of subsequent requests from these devices. In this fashion the I/O and memory busses can operate largely independently and most transactions on the general purpose bus will not suffer long delays in accessing primary memory.

2.1.5 Graphic Conventions for Bus Transactions

Two graphic techniques are employed in this thesis to describe bus transactions, especially the secure forms of these transactions developed in later chapters. The first, an *event graph*, shows the flow of data among the processing steps in the transaction and provides symbolic timing information. Event graphs indicate points in a transaction where there is potential for parallelism without making any assumptions about the performance or configuration of devices. The second, a *timing diagram*, shows the utilization of various devices during a transaction, illustrating the parallelism achieved by using a specified number of devices under stated timing assumptions. Timing diagrams are useful for determining the transaction time and cycle time of transactions for various equipment configurations.

In event graphs, processing steps are represented as labelled circles. The labels consist of a symbol to indicate the type of step and a number to distinguish among multiple instances of the same step type. Narrative descriptions of transactions refer to the steps using these labels. Table 2-3 lists the symbols used to label processing steps. (Some of these symbols refer to operations that are described later in the thesis; they can be ignored for the moment.) The flow of data (and time) is from left to right and is indicated by arcs joining process-step circles. The inputs and outputs of a transaction, as seen by the bus master, are indicated by bold dots and

SYMBOL	PROCESSING STEP DESCRIPTION
C	encryption/decryption of a 64-bit data block
T	transmission of $\leq$32 bits on the bus
A	access to read or write a memory cell
E	calculation of a 64-bit cryptographic error detection code
P	processor interrupt handling
X	XOR (modulo 2 sum) of two $\leq$32-bit quantities
=	comparison of two $\leq$32-bit bit strings

Table 2-3: Symbols Used in Event Graphs and Timing Diagrams

are accompanied by explanatory labels. The steps that comprise a bus transaction occur at three sites in the system, the current bus master, the bus and the addressed slave. To illustrate the parallelism inherent in this distributed environment, process steps are grouped along three horizontal axes corresponding to the master, bus and slave.

In timing diagrams each independent device instance, e.g., a cryptographic device or bus lines, is represented by a separate, labelled, fine horizontal line. These devices are grouped (vertically) corresponding to the event graph, i.e., bus master devices are at the top, followed by the bus and by slave devices. Time is divided

into bus-cycle duration quanta, indicated by fine vertical lines, and these lines are numbered at the bottom of the diagram. The actual duration of a bus cycle is not indicated since only relative times are needed to perform the required calculations. Cycles during which a device is busy are indicated by a bold horizontal line, labelled as in the corresponding event graph. Some events, e.g., bit string comparisons or modulo 2 addition, are not noted since they are quite fast and thus are effectively absorbed by adjacent event times. Figure 2-3 illustrates the conventions used in event graphs and timing diagrams as it describes two simple bus transactions.

Minimum transaction time (assuming maximal parallelism) is determined by the longest path in an event graph, i.e., the sum of the process-step times along that path. This time is represented as an expression in which lower case versions of process-step labels are used to subscript a time symbol (T). Thus the time to transmit 32 bits on the bus is T_t and the time for an encryption/decryption operation is T_c. Again, only major operations (those which appear in timing diagrams) are included in timing expressions. Some slight confusion arises in dealing with memory accesses in event graphs, timing diagrams and timing expressions. In timing diagrams the symbol A represents the activity of accessing memory and its duration is the *cycle time* of the memory access, but in timing expressions T_a represents the *access time* of memory. In reading a memory cell, the value is available in time T_a after the address is received even though memory is busy (unavailable) for the full cycle time. On writing a memory cell, the cycle time may begin when the address arrives, even though the data may not yet be available. The event graphs use the symbol A for both read and write accesses.

2.1.6 Standard Bus Transactions

Figures 2-3 and 2-4 provide the event graphs and timing diagrams for the three standard transactions: **read**, **write** and **interrupt**. (These transactions are referred to

as *standard* to differentiate them from the *secure* transactions developed later in the thesis.) The event graphs and timing diagrams for these transactions are fairly simple but they illustrate the basic features of both methods of graphically portraying transactions. In the timing diagrams in these figures the assumption is made that memory access time is equal to bus cycle time, i.e., fast memory is paired with a fast bus and slow memory with a slow bus. Although other combinations are possible, this convention is adopted throughout this thesis, simplifying timing calculations. However, using the event graphs and narrative descriptions provided throughout the thesis, the interested reader can construct timing diagrams for transactions under other (less convenient) relative performance characteristics.

A standard **read** begins when the bus master asserts the address of the location to be read using a **PRESENT-ADDRESS** (T1). The slave accesses the indicated location (A) and responds with the requested data using a **PRESENT-DATA** (T2). A **write** begins when the bus master asserts the address of the location to be modified, using a **PRESENT-ADDRESS** (T1), then the data is transmitted using a **PRESENT-DATA** (T2) and the slave responds immediately with an **ACKNOWLEDGE** (T3). An **interrupt** is signalled by transmitting the interrupt vector using a **PRESENT-DATA** (T1) and the processor responds with an **ACKNOWLEDGE** (T2). Processing of the interrupt (P) begins as soons as the vector arrives. The transaction time for a **read** is $2T_t + T_a$, for a **write** it is $3T_t$ and for an **interrupt** it is $2T_t$. The derivation of these timing expressions from the event graphs is straightforward and is verified by the corresponding timing diagrams. Under the relative timing assumptions noted above, **read** and **write** transactions both require 3 bus cycles and an *interrupt* requires 2 cycles. Since only one data word is transmitted every three bus cycles, the effective transfer rate of the bus is one third of its maximum potential. For busses with cycle times over the range of 100-200ns, the maximum attainable transfer rate is about 53-106 Mbits/s for these transactions.

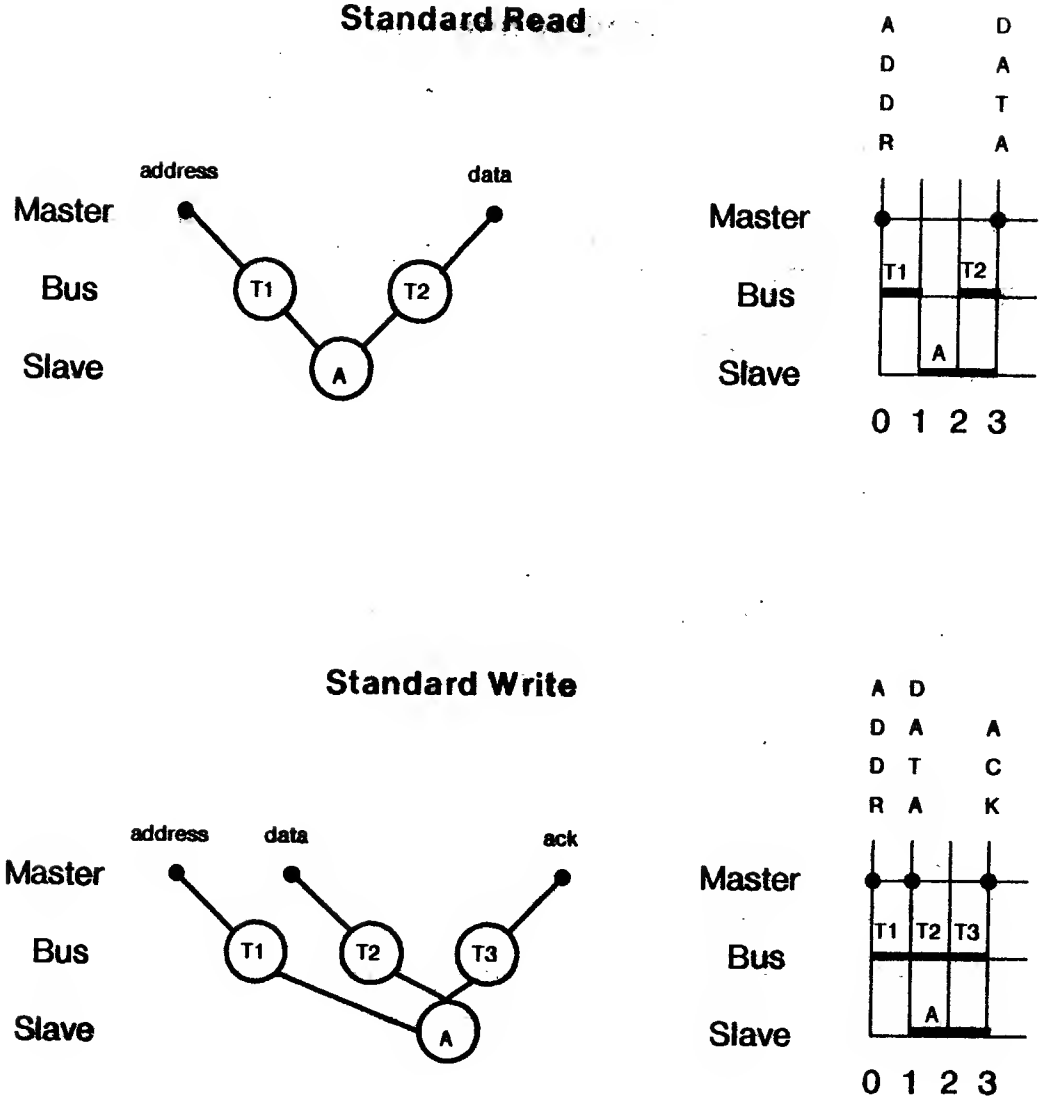


Figure 2-3: Event Graphs and Timing Diagrams for Standard **read** and **write** Transactions

For cache-equipped systems there are one or two additional transactions. Both write-through and write-back caches require **extended read** transactions but only write-back caches require **extended write** transactions. These transactions transfer an entire cache line (2, 4 or 8 words) between primary memory and the cache in one

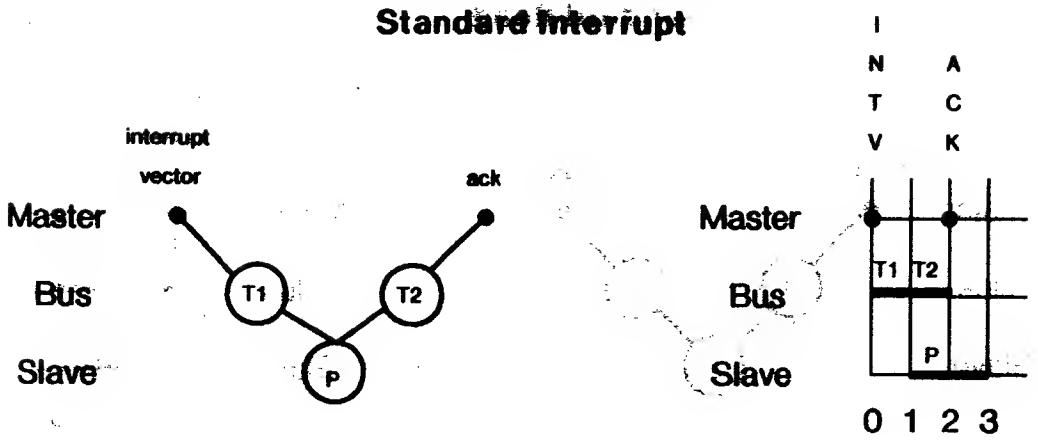


Figure 2-4: Event Graph and Timing Diagram for a Standard **interrupt** Transaction

transaction. Figure 2-5 provides the event graphs and timing diagrams for both transactions using two-word cache lines and two-way memory interleaving. An **extended read** begins by asserting the address of the word which caused the cache miss, using a **PRESENT-ADDRESS** (T1). This word is fetched first from primary memory (A1) and transmitted using a **PRESENT-DATA** (T2). The remaining words in the containing cache line are fetched (A2) and transmitted (T3) without issuing further **PRESENT-ADDRESS** operations. An **extended write** begins with a **PRESENT-ADDRESS** (T1) followed by **PRESENT-DATA** (T3,T4) operations confirmed by an **ACKNOWLEDGE** (T5). Two-word cache lines yield transaction times of $2T_t + 2T_a$ for an **extended read** and $4T_t$ for an **extended write**. Under the relative timing assumptions noted above, both transactions require 4 bus cycles to transfer two words, a bus transfer rate of 80-160 Mbits/s.

The higher bus transfer rate achieved in extended transactions comes about by eliminating explicit **PRESENT-ADDRESS** operations associated with subsequent words in the cache line. As the cache line width grows this yields even greater transfer rates. For example, a 4-word cache line can be transferred using 7 bus

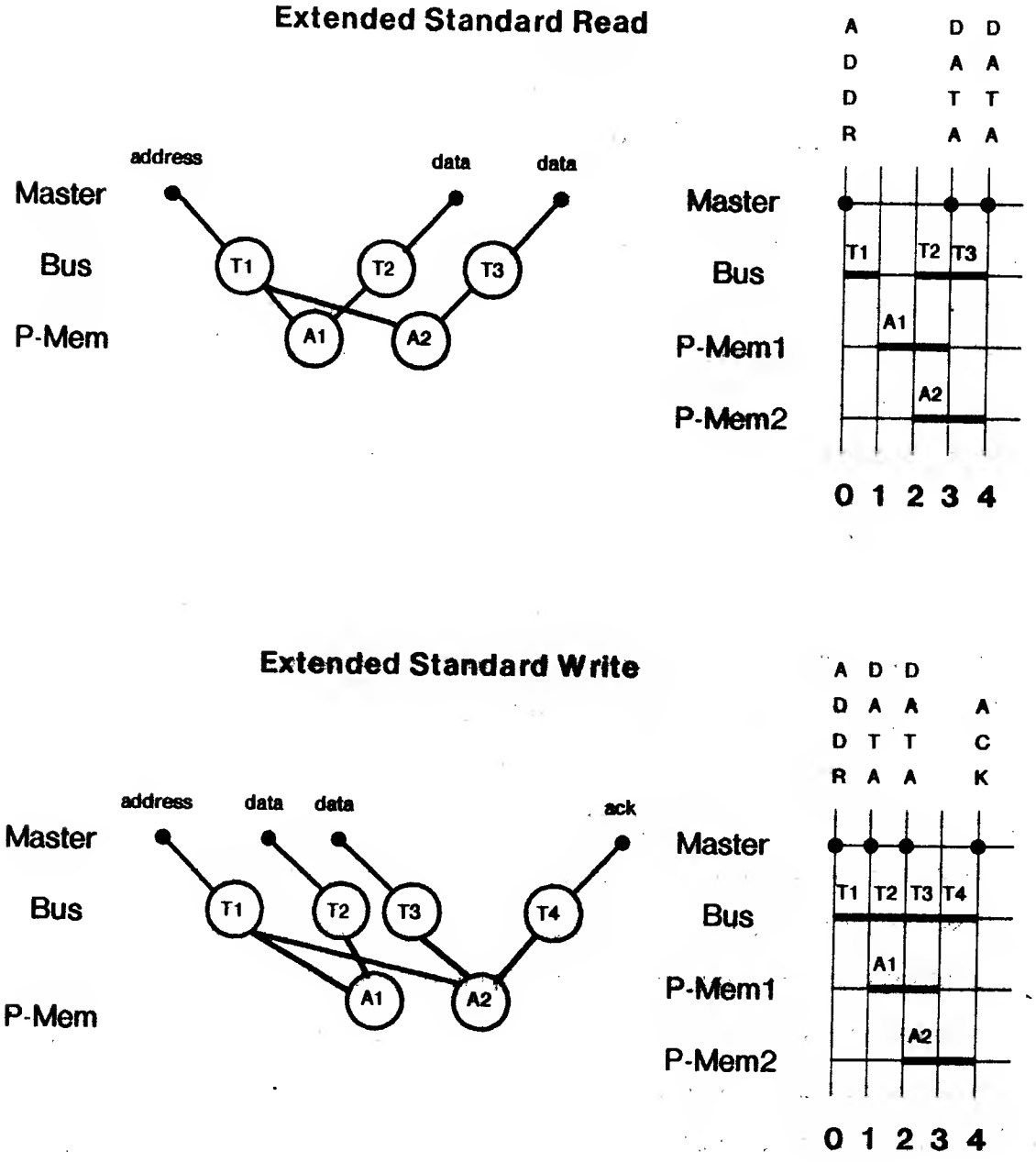


Figure 2-5: Event Graphs and Timing Diagrams for Extended Standard Transactions

cycles, a bus transfer rate of 91-183 Mbits/s. This approach to implementing extended transactions requires increased sophistication on the part of the memory controller, to generate the appropriate addresses to fetch or store each word in the cache line after the first. It is also necessary to interleave memory so that subsequent accesses can proceed without waiting for a memory access cycle to complete. Since cycle time is assumed to be about twice access time, two-way interleaving of memory is adequate for all cache line widths under this scheme. An alternative approach to implementing cache/memory transfers uses memory interleaving and additional bus lines to fetch or store multi-word units. However, the scheme adopted here should provide adequate bandwidth for the processors in the systems of interest without incurring the expense of extra bus lines.

2.1.7 Bus Utilization

Armed with the performance characteristics of various devices on the bus, one can make some rough estimates of bus utilization in the systems of interest. Precise bus utilization figures are application and equipment dependent, but even rough estimates are useful in evaluating the performance impact of the protection mechanisms proposed in subsequent chapters. (These mechanisms often increase bus utilization by "protected" devices.) In general, bus utilization in single bus, cacheless systems will be very high but can be moderated by the addition of a cache. In dual bus systems, I/O bus utilization is likely to be low but the memory bus will be very busy unless a cache is employed. In support of these statements consider the following estimates. A secondary storage device may demand up to 10-30% of the bus cycles during a transfer operation, depending on the bus speed and device transfer rate. T&A storage devices contribute somewhat less to bus demand and are used less frequently, but they can generate transient loads of 5-10%. The bus utilization of a network interface depends on network bandwidth but 10-35%

transient utilization is possible. Manipulation of images on a bit-map display can absorb essentially all of the bus cycles for short periods. Other I/O devices place only minor demands on the bus, e.g., <10% aggregate.

Bus utilization by the processor varies greatly between cache-equipped and cacheless systems. In a cacheless system, the assumption is made that the bus cycle time and primary memory access time are chosen to yield an effective memory access time equal to the minimum instruction execution time, producing a well balanced system. For example, a 100ns cycle time bus paired with a 100ns access time memory yields a system capable of supporting a processor with a minimum instruction time of 300ns (3.3 MIPS maximum). If the average time between processor-generated memory references is about 3-4 times the minimum instruction time, the processor will require about 25-33% of the bus cycles on the average with peak utilization near 100%. Using a cache with a 100ns access time, the same processor requires an average of 5%-15% of the cycles using a fast bus and memory and 10%-30% for a slow bus and memory. Of course cache misses generate transient bus utilization of 100%.

2.2 Tamper-Resistant Modules

As noted in Chapter 1, the vendors of external software have two major security requirements: preventing disclosure or redistribution and detecting modification of external software. Using the system model described in section 2.1, a number of specific attacks that violate these requirements are readily identified. The assumption is that the system components identified in Figure 2-1 are unprotected and that an attacker can examine or modify data in these unprotected components using appropriate equipment. For example, demountable media used for secondary or T&A storage can be removed from the system and the data contained therein can

be read or modified. A more sophisticated attacker might attach probes to the bus to passively or actively wiretap bus transactions, e.g., to record transmitted data or to generate spurious transactions that modify data in the system.

2.2.1 TRM Characteristics

These simple examples illustrate the need to provide some form of protection against physical tampering for those portions of the system which are critical to the secure operation of external software. At a minimum, the processor will be contained in a tamper-resistant module (TRM) since the software and databases otherwise cannot be protected during execution. A TRM has the characteristic that it prevents release or modification of the data contained therein as long as the module is intact. If a TRM is (physically) breached it is assumed that any sensitive information inside the module is destroyed (erased). If external software (including any databases critical to secure operation) is stored, executed and transferred wholly within a TRM, the security requirements of vendors can be met since disclosure and undetected modification of the software can be prevented.

The difficulty associated with engineering a TRM that performs as noted above depends on several factors. The guiding principle is that the cost of subverting the TRM should be greater than the expected gain resulting from the subversion. Thus TRM design is influenced by the value of the software being protected. The cost of subverting a TRM includes not only the price of acquiring the module and the effort involved in breaching it, but also any penalties resulting from detection of tampering. For example, if a client were to rent a TRM from a vendor and the vendor were to inspect the module and discover evidence of tampering, the vendor might refuse to furnish any other software to the client and might institute legal action against the client. Thus the cost of subverting a TRM must reflect the likelihood of detection and consequent institution of punitive measures by a vendor.

This suggests that engineering a TRM may be much easier if the TRM is not owned by the client/attacker but rather is rented from a vendor who retains the right to inspect the module and who can institute appropriate (legal) measures if evidence of tampering is discovered.

Although the details of engineering TRMs are beyond the scope of this thesis, one can make some general observations about characteristics of TRM packaging. First, it should be noted that some commercial cryptographic devices available today incorporate fundamental TRM design criteria. For example, these devices may be housed in seamless metal cases with access controlled by a pair of high security locks. These devices are designed to erase the cryptographic keys contained within whenever the device is opened, to prevent the leakage of information via electromagnetic radiation, to withstand external electromagnetic interference, etc. Although these devices are not designed to withstand a prolonged attack by a sophisticated tamperer, they do suggest that TRMs can be engineered for the level of security appropriate for commercial applications.

One of the most important characteristics of a TRM is its ability to destroy sensitive data contained within should it detect any evidence of tampering. This destruction of data must be carried out quickly to prevent a would-be tamperer from accessing the information after breaching the TRM. Rapid erasure of a large quantity of non-volatile memory, e.g., in secondary or T&A storage devices, may prove difficult or impossible depending on the storage technology employed. Thus magnetic bubble memories might provide an attractive form of secondary storage for TRM packaging while media such as disks may be less well suited to this application.

Another aspect of TRMs that must be noted is their impact on flexibility of system configuration. In configuring a computer system composed of one or more TRMs, the user will probably be restricted in the selection of components. In part

this restriction arises because not all devices or combinations of devices are amenable to TRM packaging. Moreover, all devices in a TRM (or a collection of co-operating TRMs) must be packaged by the vendor of the system since all of these devices must perform correctly to maintain the security of the external software. This requirement may result in some combinations of devices being unavailable as a TRM-packaged system. The ability to expand a system may be hampered by lack of space within a TRM to incorporate more components. Maintenance of TRM-packaged devices is hampered since only the TRM vendor is in a position to provide service while maintaining system integrity.

An important consequence of TRM packaging is the cost incurred. Packaging one or more devices as a TRM is more expensive than standard (non-secure) packaging. Although the differential in cost between standard and TRM packaging varies based on the perceived threat environment, experience in packaging commercial cryptographic devices indicates that this cost can be quite substantial. For example, the difference in price between one conventionally packaged (rack mount) link encryption device and the same device packaged for use in unsecure areas (desk top box) is approximately $900, roughly 45% of the total price of the latter unit. It appears that the majority of this cost arises not from additional electronic components but from mechanical engineering considerations. Over and above some base, the cost of building a TRM probably increases with the size of the TRM, for a fixed level of security. Thus very large TRMs may be impractical because the cost of packaging would be great and very small TRMs may be infeasible because the cost of packaging would be significantly greater than the cost of the protected components. Only over some middle range is TRM packaging likely to be practical.

It may be cheaper to build a TRM that is permanently sealed, as opposed to one that includes provisions for controlled access, and the resulting device may be more

secure. The assumption here is that provisions for controlled entry into the module introduce weak points that must be buttressed by sophisticated and costly security mechanisms. It may also be easier to detect tampering in permanently sealed modules. TRMs sealed at the time of manufacture would include no provision for controlled access for maintenance, thus eliminating the need for trusted field service personnel. If a component within a sealed TRM fails, the entire TRM would be replaced and the failed TRM would require "factory" servicing and re-packaging (the contents would be erased during servicing). This approach to TRM packaging would probably work well with devices that are highly reliable, e.g., solid state devices, but not with electromechanical devices that require periodic servicing. Sealing a TRM eliminates the option for field upgrades or expansion. Finally, the number of components that can be packaged in a sealed TRM is limited by the fact that the failure of any component may require replacement of the entire TRM.

2.2.2 A Monolithic TRM Approach

As a first approximation to protecting external software, one could imagine enclosing all of the devices that are critical to the secure operation of the external software in a monolithic TRM, as illustrated in Figure 2-6. (The specific system configuration used within the TRM is not important here since all of the security relevant components are entirely within the TRM.) The security requirements of a vendor can be met by this sort of system since the processor, all storage required by external software and the bus connecting these devices are all contained within the TRM. Note that not all of the system components are enclosed in the TRM. Terminals and other peripheral devices that do not effect the secure operation of external software can be attached to the bus outside of the TRM. Even storage devices for data not essential to the secure operation of external software could be attached to this bus extension, e.g., secondary storage exclusively for client data

could be provided outside the TRM. In order to attach other devices to the bus without violating the security provided by the TRM, the bus extension requires a special *secure bus coupler* (SBC).

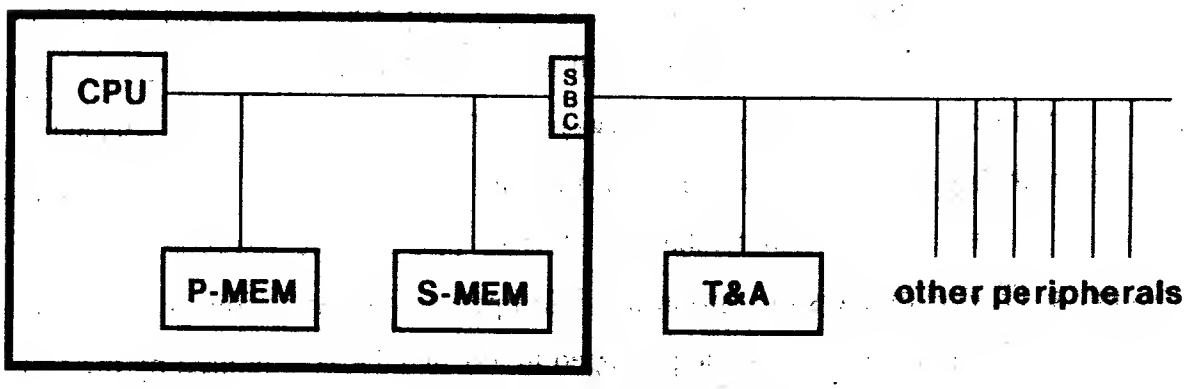


Figure 2-6: Using a Single TRM to Protect a System

The SBC acts as a filter to prevent unauthorized disclosure or modification of data within the TRM. To this end, the SBC ensures that bus traffic among devices within the TRM is not repeated onto the bus extension (to prevent disclosure) and it controls access to primary memory by DMA devices outside the TRM (to prevent disclosure and modification). These tasks are made easier by partitioning the bus address space so that a single address line indicates whether an addressed device is inside or outside the TRM. It then becomes trivial for the SBC to avoid repeating intra-TRM bus traffic onto the bus extension by inspection of this address line. To control access by DMA devices to primary memory, the processor must inform the SBC of the locations that should be accessible to DMA devices outside the TRM, along with the mode of access allowed, i.e., read or write. The SBC can be equipped with a small number of registers to establish the bounds and access modes for these

locations. These registers are managed by the processor as part of controlling "unsecure" DMA devices[5] and are scanned on transactions initiated outside the TRM.

This approach to securing external software has several advantages. Little in the way of special hardware is required, only the SBC is unique to the design, the remaining devices can be "off the shelf." The SBC appears relatively easy to construct and should be capable of operation at bus speeds, given the existence of analogous devices such the the UNIBUS adaptor employed on the VAX 11/780 [10]. The only impact on software is the requirement to co-ordinate management of the SBC with control of DMA devices on the bus extension, a function easily assumed by the operating system as part of device management. The design also provides some flexibility in system configuration. For example, secondary storage for client files might be provided on devices attached to the bus extension whereas secondary storage for external software is provided by devices within the TRM. Despite the advantages noted above, this design also has a number of drawbacks.

Perhaps the most obvious problem with this design is that it does not provide for demountable secure storage. Thus no secure T&A storage can be provided, as noted by its absence from the TRM in Figure 2-6, and secondary storage contained in the TRM cannot employ demountable media. The lack of secure transfer storage could be a major problem if the only alternative were the use of erasable PROM (EPROM) or factory-recorded secondary storage within the TRM. Note that ROM is not acceptable for recording external software because of the need to be able to erase the sensitive information contained in the TRM in case of tampering.

---

[5]For the SBC to be completely transparent, it would have to be aware of the addresses and semantics of the control registers for all of the devices on the bus extension. This would significantly complicate the SBC and would limit the choices for devices on the bus extension to those with which the SBC was familiar. For these reasons a transparent SBC design was rejected.

Similarly, only readily erased devices such as bubble memories are suitable for inclusion as pre-recorded secondary storage. Factory recording of external software is not very appealing as it does not support distribution of new releases, either for bug fixes or new products.

However, secure distribution of external subsystems can be provided using communication facilities and employing cryptographic techniques as described in the next section. Using such techniques, the vendor can securely transmit copies of or updates to external software to appropriately equipped, TRM-packaged computer systems. Thus the lack of secure transfer storage can be overcome, at the cost of requiring some communication facilities and cryptographic capabilities within the TRM. Whether the inability to provide demountable secure storage for non-transfer purposes is a serious deficiency depends on the applications involved. For example, an external subsystem that managed client databases using data structures and access techniques that were viewed as proprietary might require secure demountable media for secondary or archival storage. The inability to provide secure demountable media for secondary or archival storage is a serious limitation in some applications.

Another difficulty with this design is that it may encounter the erasure problem alluded to earlier, because of the presence of secondary storage within the TRM. Again, the seriousness of this problem will depend on the volume of non-volatile memory contained in the TRM and the technology used to implement it. Although this design exhibits some flexibility in allowing a user to configure a system with non-security relevant devices outside the TRM, in other ways the design allows little flexibility. As noted earlier, the users may be quite limited in their choice of configurations for devices within a TRM, and in this design most of the system is within the TRM. Since secure secondary storage is available only within the TRM, some types of storage devices may be precluded because of size constraints or

because of the need for periodic adjustment. The number of devices contained in the TRM probably rules out use of the sealed TRM packaging technique described earlier and for some systems the size of the TRM required would pose a significant expense.

The impact of these characteristics on system design are illustrated in the following examples. One sort of system that might be amenable to the monolithic TRM design is a very simple personal computer designed exclusively for running a language system such as BASIC or APL. The TRM could contain the language system in EPROM or bubble memory and an amount of primary memory suitable for simple applications could be provided. Secondary memory within the TRM might not be required, making a small, sealed TRM a real possibility. User programs and data could be kept in a secondary storage device attached to the bus extension, along with a terminal and other input/output devices. If the only external software to be protected were the language facilities, and if these facilities did not require distribution of new releases to fix bugs or to add enhancements, this design might prove adequate. To accommodate a more flexible update strategy, a cryptographic device, a facility for re-writing the EPROM or bubble memory and some communication capability could be included to support remote updating.

One can imagine a number of variations on this simple scenario that highlight the deficiencies of the monolithic TRM design. For example, if the vendor of the personal computer wanted to sell proprietary application software to his clients, secure secondary storage within the TRM would be required and the problems of providing such storage within the design have been pointed out above. These problems also arise if the vendor requires the object code produced by the language system to be protected from disclosure, in order to hide the code generation techniques employed. Similar problems arise in the context of nodes in a distributed system. For example, a secure database residing at a node would have to

be contained in secondary storage within the TRM and here the lack of demountable storage and the problems of large quantities of non-volatile memory within a TRM essentially preclude use of this design. Thus this design is inadequate for many classes of applications.

2.3 Cryptographic Terminology, Concepts and Techniques

Cryptographic techniques are used in four distinct contexts in this thesis. Network-based distribution of external software requires secure communication between a vendor and his TRMs. This method of software distribution is critical to the monolithic TRM approach, since that approach does not support secure T&A storage, and it may be the preferred distribution method for the other design approaches as well. This section presents the basic communication security techniques necessary for secure, network-based distribution of external software. The encrypted bus approach examined in Chapter 3 relies on secure communication among TRMs connected via a physically unprotected bus. That chapter presents modified communication security techniques for this highly specialized communication environment (the bus). The encrypted storage approach of Chapter 4 develops special cryptographic techniques to protect data stored outside a TRM. Finally, in Chapter 5, cryptographic techniques and protocols are used to distribute external software to TRMs provided by third-party suppliers. This chapter is not a general tutorial on cryptography; it merely attempts to provide some background necessary to understand the cryptographic techniques employed in subsequent chapters.

2.3.1 Terminology and Basic Concepts

A *cryptographic algorithm* or *cipher* is an algorithmic transformation performed on data on a symbol-by-symbol basis. In *enciphering* or *encrypting* data, the *plaintext* input is transformed into unintelligible *ciphertext* output. The inverse of this operation is referred to as *decryption* or *deciphering* and it transforms ciphertext into the plaintext from which it was derived [32]. These transformations are carried out under the control of a *key*. In *conventional ciphers* (CCs) such as the NBS Data Encryption Standard (DES) [23], the same key is used for enciphering and deciphering a collection of data. On the other hand, *public-key ciphers* (PKCs) such as the RSA algorithm [26] use different, but mathematically related, keys for encryption and decryption. These terms are illustrated in Figure 2-7.

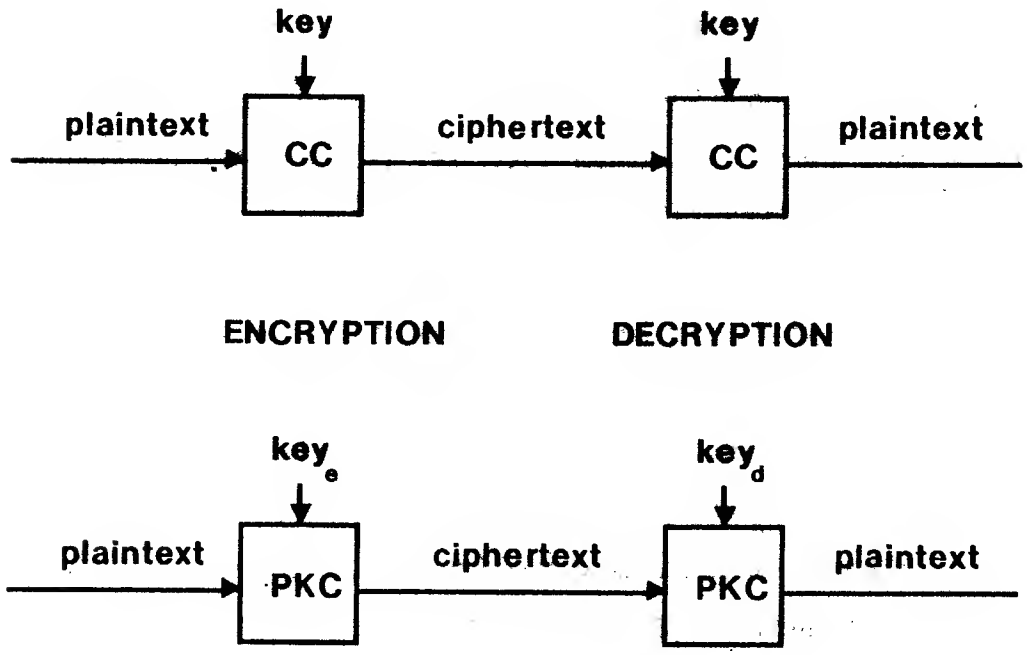


Figure 2-7: Conventional and Public-Key Cipher Configurations

For both conventional and public-key ciphers the assumption is made that the algorithm is known not only to the users of the cipher but also to any attackers. The secrecy, *authenticity* and *integrity* guarantees[6] accorded data transformed by these ciphers derive from their mathematical structure and from the secrecy of keys used to parameterize the ciphers. In conventional ciphers, an attacker cannot decipher ciphertext nor can he generate ciphertext that will decipher into predictable plaintext without knowledge of the key used to generate the ciphertext. Thus, in these ciphers, the secrecy of the key provides concealment and the basis for determining the authenticity and integrity of ciphertext. In public-key ciphers, the key used to encipher data ($\mathbf{key}_e$) need not be kept secret in order to effect concealment integrity checking. This is because a different key ($\mathbf{key}_d$), related to the encryption key in a complex fashion, is used for decryption. Because of the mathematical structure of public-key ciphers, knowledge of $\mathbf{key}_e$ does not allow a cryptanalyst to determine $\mathbf{key}_d$.

This property of public-key ciphers decouples secrecy from authenticity and integrity. Data transformed under PKC key ($\mathbf{key}_e$) carries no guarantee of authenticity since this key is usually publicly available and thus anyone can encipher data using it. Moreover, only the holder of the matching decryption key ($\mathbf{key}_d$) can decipher data encrypted under $\mathbf{key}_e$, so this scheme provides secrecy. Conversely, data transformed under $\mathbf{key}_d$ can be deciphered by everyone, since $\mathbf{key}_e$ is public, but such data can be verified as authentic and its integrity can be checked because only the holder of $\mathbf{key}_d$ can generate ciphertext that is predictably decipherable under $\mathbf{key}_e$. (Despite designations as *enciphering* and *deciphering* keys, both PKC keys transform plaintext to ciphertext and invert the transformation performed by the complementary key.) Thus transformation under a public key provides secrecy

[6] In this context, data is considered *authentic* if it was enciphered by an authorized party and its *integrity* has not been violated if the ciphertext has not be modified.

whereas transformation under a secret PKC key provides a basis for authenticity and integrity checking.

In communication contexts, a PKC key pair is associated with each user. Secret, authentic, integrity-checked communication between two users can be achieved by transforming each message twice at the transmitter and at the receiver, as illustrated in Figure 2-8. The transmitter first transforms the message under his secret key (**T-key$_d$**), for authenticity, and then under the public key of the intended receiver (**R-key$_e$**), for secrecy. (Both transformations contribute to the integrity guarantee.) Upon receipt of the message, the receiver transforms the message under his secret key (**R-key$_d$**), then under the public key of the transmitter (**T-key$_e$**), to reveal the original plaintext. Of course, the secrecy, authenticity and integrity guarantees provided by these transformations are valid only if both transmitter and receiver are correctly informed as to each other's public keys.

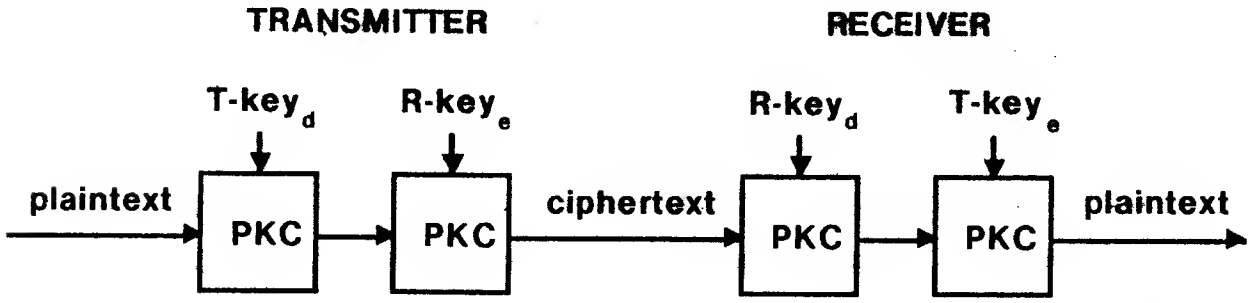


Figure 2-8: Providing Secrecy, Authenticity and Integrity with Public-Key Ciphers

Even though public-key ciphers provide some features not available in conventional ciphers, the former are not well suited to most of the applications in this thesis. For example, public-key ciphers offer some potential advantages over conventional ciphers in distributing cryptographic keys. The first three applications

of cryptography in this thesis, as noted at the beginning of section 2.3, do not encounter complicated key distribution problems and would not benefit from the use of public-key ciphers. Thus almost all of the techniques employed in this thesis are based on conventional ciphers and public-key ciphers are employed only in some applications in Chapter 5. In fact, public-key ciphers are immediately eliminated from consideration for most of these applications because of the relatively low throughput achieved by their implementations, as described in section 2.3.5.

Good ciphers, both conventional and public-key, exhibit high resistance to a variety of cryptanalytic attacks. Obviously ciphers must resist attempts by attackers to determine the key required to decrypt a quantity of ciphertext or to discover the plaintext from which the ciphertext is derived through examination of the ciphertext (*ciphertext only attack*). Moreover, an attacker should not be able to deduce the key used to decipher data even if he is given matching plaintext and ciphertext (*known plaintext attack*). The same holds true if the attacker is given the opportunity to select the plaintext for which matching ciphertext is made available (*chosen plaintext attack*). These requirements are motivated by the fact that an attacker will often be able to know or to choose some plaintext that will be encrypted and become available to him as ciphertext. For example, in the context of protecting external software, one might encounter enciphered relocatable program files, portions of which are likely to contain easily predicted values. In the same context, an attacker might be able to choose values that would become part of an encrypted database, providing a chosen plaintext attack.

The ciphers selected for use in this thesis, the DES and the RSA algorithm are designed to resist the cryptanalytic attacks described above. Nonetheless, one must exercise care in using these ciphers or subtle weaknesses may arise. For example, not all cryptographic techniques automatically compensate for plaintext that varies

over a very small range of possible values or plaintext that contains recurring patterns. Unless suitable precautions are taken, these plaintext characteristics may be visible in the ciphertext, resulting in information disclosure. Techniques for verifying the authenticity and integrity of encrypted data in the face of attacks often rely on the presence of predictable information in plaintext and on error propagation characteristics of ciphers. Since the plaintext encountered in this thesis may admit to a wide range of values, predictable information must be supplied explicitly for security purposes. Different ways of using ciphers yield different error propagation characteristics and this must be considered in designing mechanisms for checking authenticity and integrity of data. The following sections describe specific techniques for preventing disclosure and detecting modification.

2.3.2 Block Cipher Techniques

Most modern cryptographic algorithms (conventional and public-key) are block ciphers, i.e., they operate on fixed-size blocks of plaintext and ciphertext. For example, the block size of the DES is 64 bits and for the RSA algorithm a block size of about 320 bits yields comparable security. The simplest way of using a block cipher is sometimes referred to as the *electronic code book* (ECB) mode [16], indicating the analogy to manual cryptographic procedures, and is illustrated in Figure 2-9. (This and subsequent illustrations omit keys for clarity.) However, this mode exhibits several shortcomings. If data to be enciphered is smaller than the block size of the cipher, the data must be padded to produce a full size block. Similarly, the entire resulting ciphertext block must be presented for decryption, i.e., it is not possible to decipher a partial block. If the data to be encrypted is longer than a block it must be broken into block-size pieces and each piece enciphered separately. This mismatch between the granularity of encryption and the size of plaintext results in waste, e.g., on average half of each block may be wasted due to this mismatch.

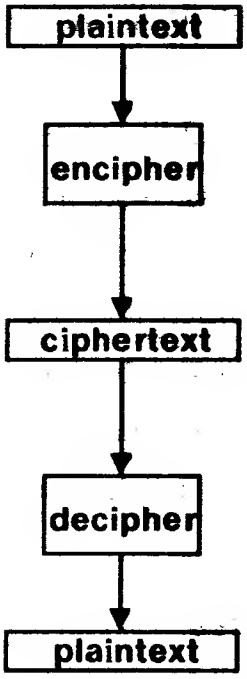


Figure 2-9: Electronic Code Book Mode for Block Ciphers

With respect to concealment, ECB mode has an obvious deficiency, i.e., identical plaintext blocks are transformed into identical ciphertext blocks. Thus plaintext patterns that occur aligned on block boundaries are visible in the resulting ciphertext. In the case of the DES, if plaintext, when divided into 8-byte blocks, exhibits block-size patterns, then these patterns will be visible in the resulting ciphertext. Moreover, if the bit pattern used to pad short blocks is constant, an attacker might be able to perform frequency analysis on the ciphertext blocks to discover the plaintext. For example, if 32-bit words are enciphered individually and each is padded with the same bit string, the resulting ciphertext blocks will vary only over the range of values assumed by the 32-bit words, and this may be small enough

to allow effective frequency analysis by an attacker. Because of these deficiencies, ECB mode is usually employed only for tasks such as distribution of cryptographic keys, where the data is random and well matched to the block size.

These concealment problems can be solved by including in each plaintext block a non-secret, unique bit string, a quantity designated as an *(in-block) initialization vector* (IV), illustrated in Figure 2-10. (The term *initialization vector* is often used in a more restricted sense in cryptography but it serves essentially the same function as the quantity described here.) The inclusion of this bit string makes each plaintext block different and thus each resulting ciphertext block is different, effectively concealing patterns and compensating for limited range plaintext, e.g., short blocks. This technique works since, in the DES, two plaintext blocks that differ by as little as one bit yield ciphertext blocks that differ in approximately 50% of the bit locations. This technique suffers from the drawback that a portion of each block must be reserved for this unique bit string, thus reducing available bandwidth in communication applications or wasting space in storage applications. However, if an application already requires inclusion of a unique bit string as part of each plaintext block, e.g., sequence numbers in a communication application, this bit string can serve as an IV so no additional space is wasted.

An alternative technique for combatting the same problem involves combining each plaintext block with a (block size) initialization vector, via modulo 2 addition, before enciphering the block. This *additive* technique is not quite so secure as the inclusion of an in-block IV since duplicate ciphertext blocks may result, providing cryptanalytic opportunities for an attacker. For example, if two ciphertext blocks are identical under this scheme, an attacker can work backwards from a knowledge of the IVs to determine the sum of the plaintext blocks. If he has knowledge of some of the plaintext in one of the blocks he can determine the value of corresponding bits in the other block. If the range of the IVs is suitably large (say 64

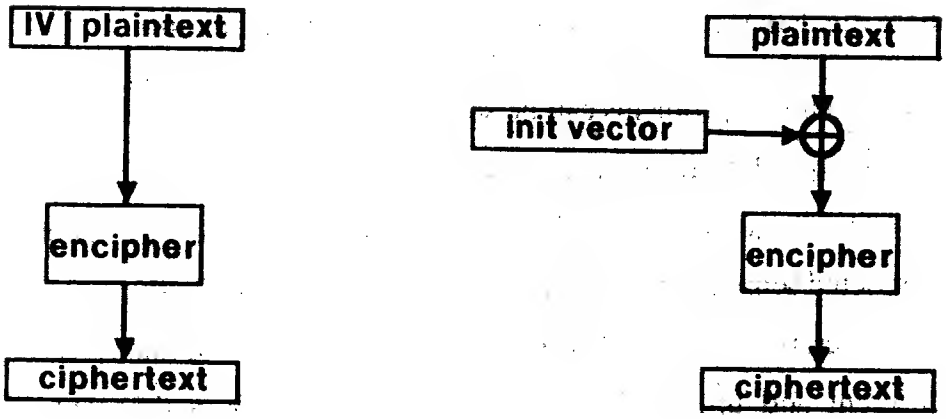


Figure 2-10: In-block and Additive Initialization Vector Techniques

bits), and the IVs are chosen pseudo-randomly, this method offers adequate security since the likelihood of duplicate ciphertext blocks is quite small. The advantage of this approach is that the IVs take up no space in the blocks, but it is necessary to know the IV associated with a block for decryption. The values of the IVs must be implicitly derived from some contextual information if there is to be any space saving. For example, in a communication application the sequence number implicitly associated with each transmitted block could serve as an IV.

The inclusion of a *predictable* quantity in each block provides a basis for checking the authenticity and integrity of the block. The object here is to verify that the block was encrypted by an authorized individual and that it has not been modified in any way after being encrypted. For a block cipher such as the DES, modification of as little as one bit in a ciphertext block results in changes to approximately 50% of the plaintext upon decryption. The same error propagation effect occurs if a ciphertext block is deciphered under a key that differs by as little as one bit from the key used to encipher the block. Thus, the inclusion of a predictable n-bit field in a plaintext

block provides a check on the authenticity and integrity of the block which an attacker can subvert with a probability of 2^{-n}. This is the probability that the n-bit field is unchanged if the ciphertext block was modified or if it was encrypted under a key other than the key used to decipher it. Such a quantity will be referred to as an *authenticity/integrity check field* (AICF).

Any predictable quantity can be included in each block as an AICF, e.g., a constant bit string. However, the functions of an AICF and an IV can be combined into a single field, reducing the space overhead that would result if an in-block IV and a separate AICF were employed. Since a combined IV/AICF field must be large enough to uniquely identify each block and large enough to detect spurious or modified blocks this may not be the most space efficient technique. For example, if the size of the IV required to uniquely identify each block is larger than the size of the AICF required to detect modification, then an implicit IV and a dedicated AICF could waste less space. Despite this ability to combine both functions in a single field, the percentage of each block devoted to such a field can be significant, especially if the block size is small. For example, in many applications a 16-bit AICF may be adequate, i.e., an attacker is allowed a 2^{-16} chance of undetectably violating the authenticity and integrity guarantee provided by the AICF. But in a 64-bit DES block this 16-bit field represents 25% overhead. One could reduce the percentage overhead by using a cipher with a larger block size, but if the application normally generates plaintext smaller than this block size, waste will result from the occurrence of partially filled blocks.

One can reduce the percentage of space devoted to security measures through *block chaining* encryption techniques. Block chaining techniques encrypt plaintext of variable lengths (integral multiples of the block size) using some form of feedback to cryptographically relate the resulting ciphertext blocks. There are a number of options for feedback mechanisms; the method described below (and

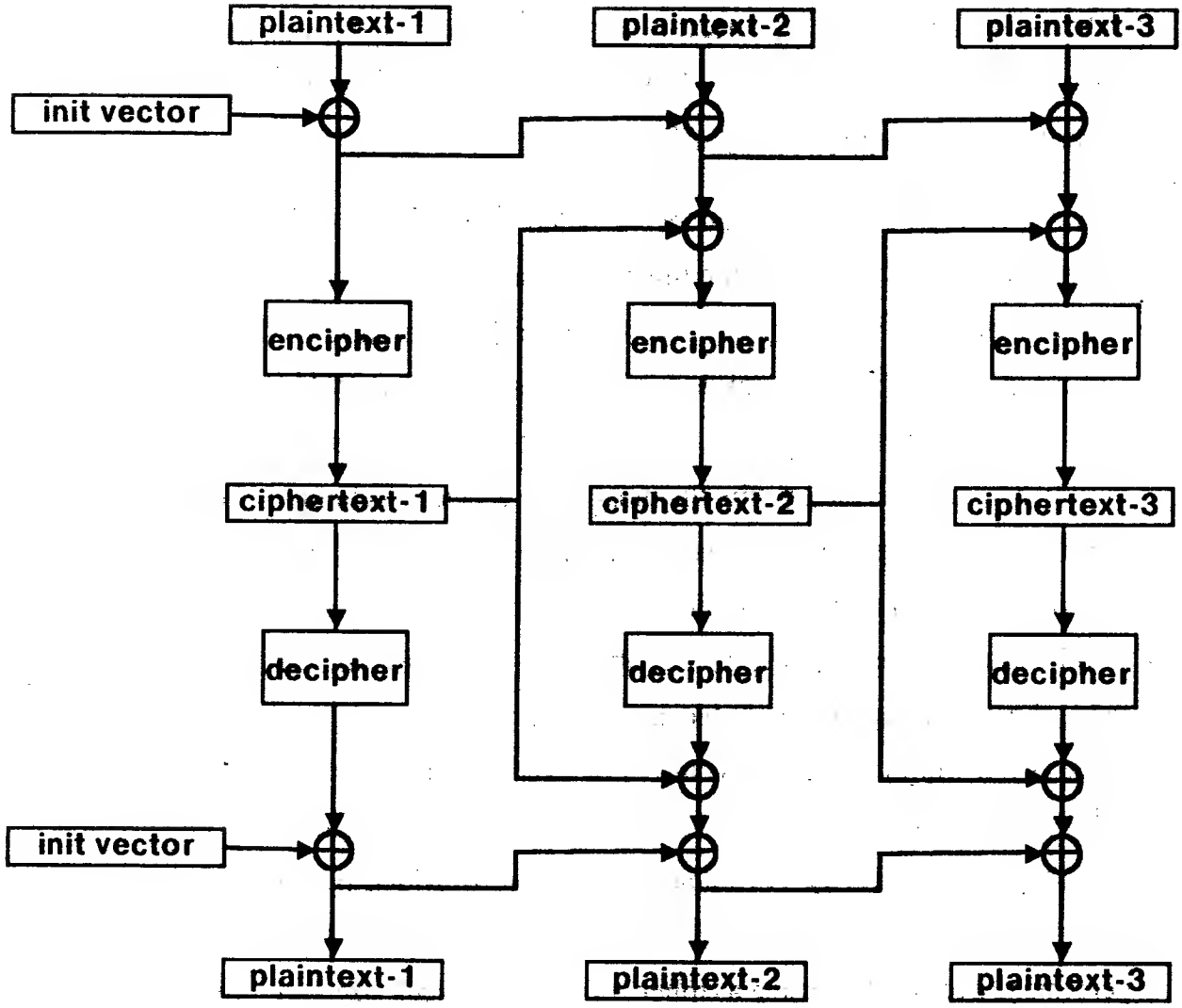


Figure 2-11: Plaintext-Ciphertext Block Chaining (PCBC)

later employed in Chapter 4) uses both plaintext and ciphertext feedback and is designated as *plaintext-ciphertext block chaining* (PCBC) [12]. In this method, the first block in the *plaintext chain* is added (modulo 2) to a block-size IV and the result is encrypted. Each subsequent block in the plaintext chain is added to the

employed since the error propagation required by an AICF is not present. However, CBC mode is somewhat simpler than PCBC mode and when used with an EDC it provides adequate authenticity and integrity guarantees. (The EDC is adequate in this case since an attacker cannot predictably modify the enciphered plaintext or the EDC.) This mode is often proposed for communication applications [16]. Block chaining based on plaintext feedback alone is generally unacceptable, since plaintext patterns may not be effectively masked, even though this mode does provide forward error propagation.

2.3.3 Stream Cipher Techniques

The cryptographic modes described above do not accommodate plaintext that is not an integral multiple of the cipher block size without waste. The 64-bit block size of the DES is well suited to most of the applications in this thesis since two 32-bit words fit into a DES block. Much of the plaintext to be encrypted is an even number of words long and for large data structures or long messages wasting half a block (32 bits) is usually not a serious problem. However, when plaintext is sub-block size, e.g., a 32-bit word, this level of waste poses a serious concern. To solve this problem, block ciphers can be used as *stream ciphers* that encrypt plaintext strings of any size. The central concept is to use the block cipher to generate blocks of pseudo-random bits, referred to as a *cryptographic bit stream*, portions of which are added to the plaintext to conceal it. (Because the cryptographic strength of this technique is based on the secrecy of this bit stream, PKCs cannot be applied here directly unless they are used as CCs, i.e., with no public knowledge of the key used to generate the cryptographic bit stream.)

There are a number of ways to generate a cryptographic bit stream using a block cipher, just as there are several choices for feedback in the block chaining modes described in the preceding section. For example, in what is often viewed as the

simplest form of stream cipher, an *autokey* cipher [32], bit stream generation begins by enciphering an IV. The resulting crypto bit stream is added to plaintext, to encipher it, and is fed back as input to the cipher to generate further crypto bit stream, as illustrated in Figure 2-12. Decryption is identical to encryption, i.e., the same crypto bit stream is added to the ciphertext to yield plaintext. Plaintext of any size can be accommodated by this cipher, e.g., by selecting a fixed portion (a bit or a byte) of each crypto bit stream block to combine with the plaintext and discarding the remainder. Of course, discarding a portion of the bit stream causes the performance of the cipher to suffer, e.g., Figure 2-12 shows only one-fourth of each block being used so the cipher runs at one-fourth of its maximum rate.

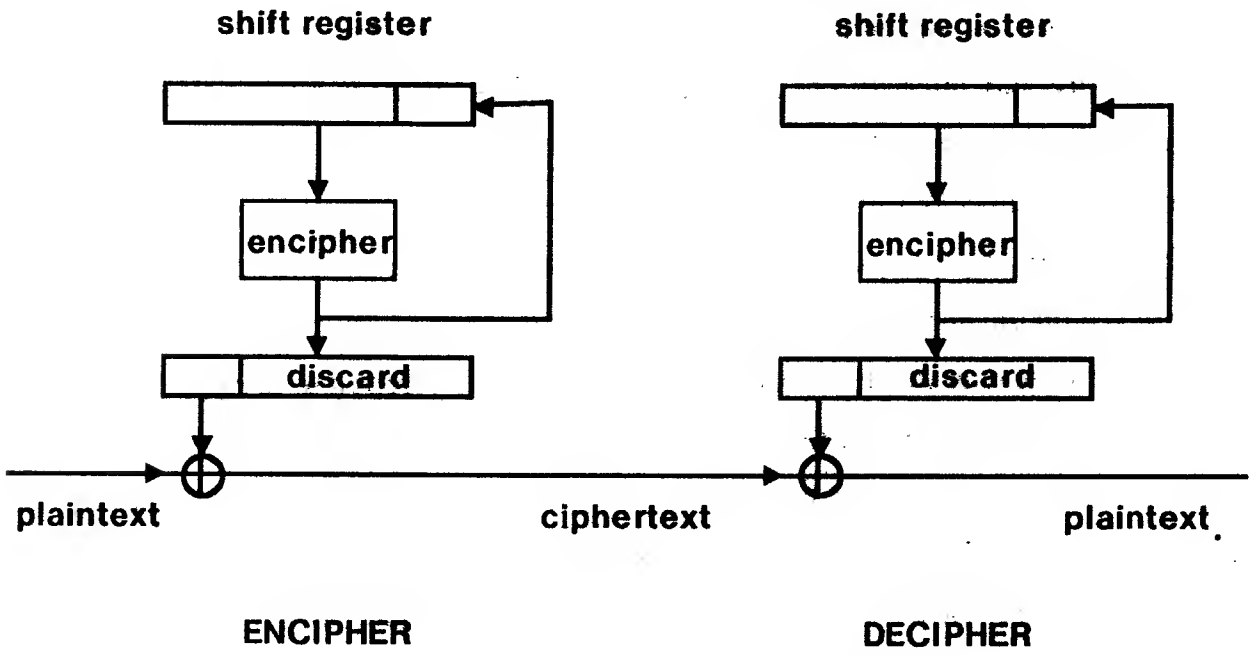


Figure 2-12: Autokey Stream Cipher Example

Depending on the application, the crypto bit stream may be generated continuously or it can be "re-initialized" periodically with a unique IV. For example, in some communication applications a continuous bit stream is transmitted to conceal all message traffic (or the lack thereof) whereas in other applications a new IV is used for each message. Note that the IVs must be unique since they determine the crypto bit stream, and if two messages were enciphered using the same IV (bit stream), an attacker could add the messages on a bit-by-bit basis to yield the sum of the plaintext. A striking feature of this stream cipher is that it provides no error propagation, i.e., if a bit of ciphertext is complemented, the corresponding plaintext bit is complemented, but no other plaintext bits are affected. (However, if a bit of ciphertext is lost, the decrypted plaintext will be garbled due to shifting over of the crypto bit stream before addition.) Thus neither an AICF nor a conventional EDC can be used with this stream cipher for authenticity and integrity checking due to this lack of error propagation. (An attacker, knowing what kind of EDC is employed, can modify the plaintext in a fashion that is invariant under that EDC algorithm.)

However, a *cryptographic error detection code* (CEDC), a cryptographic function calculated on the plaintext, can be employed to detect modification. (A CEDC used to authenticate data which is not encrypted is sometimes referred to as *cryptographic check digits* [4].) Since a CEDC is a complex function of the plaintext on which it is calculated and on the secret key used in the calculation, an attacker cannot modify the plaintext in a fashion which is invariant under the CEDC. (An n-bit CEDC, like an n-bit AICF, allows an attacker a 2^{-n} chance of undetectably modifying the covered plaintext.) A CEDC can be calculated in a number of ways. For example, a block chaining mode like PCBC or CBC can be used to encrypt the plaintext (padded if necessary to be an integral number of blocks long) and a portion of the last ciphertext block generated in this fashion can serve as a CEDC (since it is a cryptographic function of all the preceding plaintext). The other stream cipher

mode described below also may be used to generate a CEDC. Thus the lack of error propagation in an autokey stream cipher does not preclude its use where authenticity and integrity guarantees are required. However, providing these guarantees requires additional operations which may translate into reduced throughput or additional hardware.

Another stream cipher, *cipher feedback* mode (CFB) [16], is illustrated in Figure 2-13. To begin, a block-size IV is input to the cipher and encrypted to generate a cryptographic bit stream block. The plaintext is added to this bit stream and the resulting ciphertext is shifted into the cipher input and encrypted to generate the next crypto bit stream block. If plaintext is supplied in sub-block size *quanta*, e.g., bytes or bits, then a corresponding portion of the crypto bit stream is used and the remainder of each block is discarded, as in the autokey cipher described above. This process is repeated until no more plaintext remains to be encrypted. Decryption is accomplished by a symmetric, but not identical, procedure, i.e., generating the same crypto bit stream and adding it to the ciphertext to produce the plaintext. Figure 2-13 illustrates CFB mode encryption and decryption applied to plaintext quanta that are one-fourth block size.

In CFB mode, as in autokey mode, it is essential that each plaintext chain be enciphered using a different IV. Since the crypto bit stream is a function of both the IV and the plaintext in CFB mode, using the same IV on two plaintext chains results in duplicate crypto bit stream only as long as the plaintext chains are identical. Nonetheless, to avoid exposing any data, the IVs should be unique for each independently encrypted chain. As before, the IV may be implicitly derived or may be carried with each chain. This mode provides excellent concealment of plaintext patterns but the error propagation is limited. This stream cipher mode exhibits error propagation analogous to CBC mode. If a bit of ciphertext is complemented, the corresponding plaintext bit is complemented but subsequent quanta of plaintext

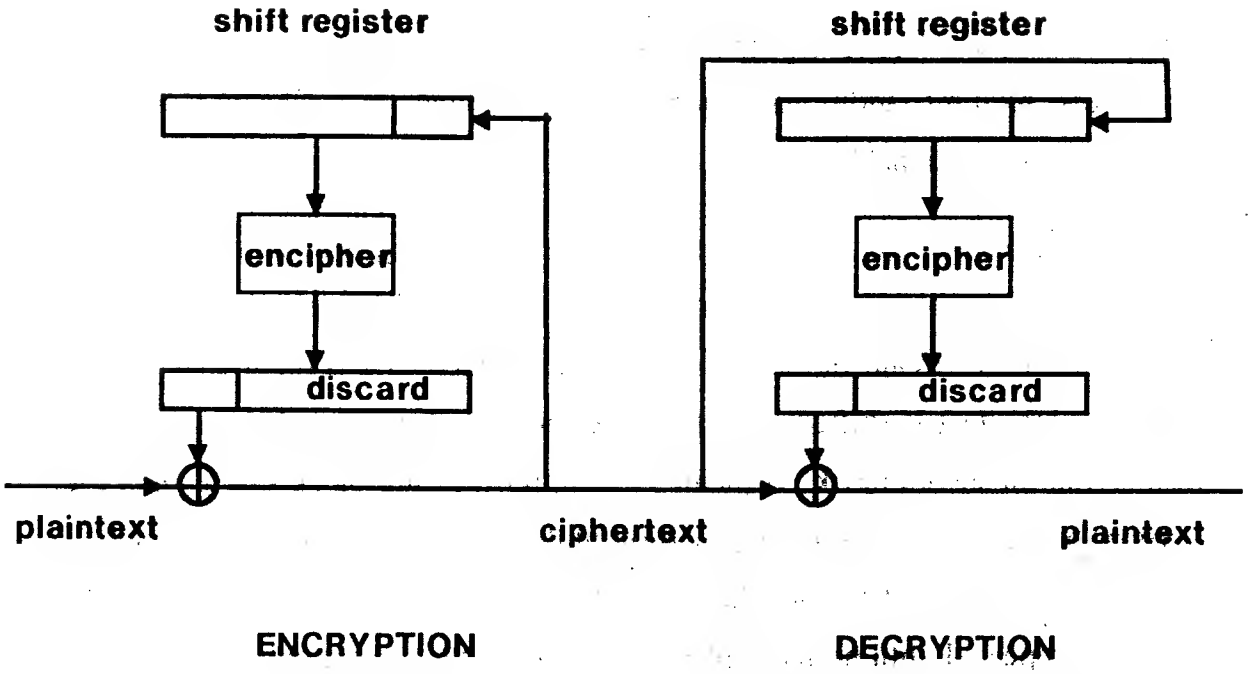


Figure 2-13: Cipher Feedback Mode Stream Cipher

are unpredictably garbled until the input shift register is cleared of erroneous ciphertext. For the DES, the shift register is 64 bits long and thus error propagation affects 64 bits of plaintext following the quanta containing the error. This error propagation characteristic means that the final enciphered quanta of plaintext in a chain exhibits no error propagation at all. Some other stream cipher modes can offer forward error propagation, but all suffer from the defect that the final plaintext quanta in a chain exhibits no error propagation.

Since the last quanta in a chain can be modified with predictable effects, one cannot place an EDC or AICF and data it is protecting in this quanta. (An attacker might be able to modify the data in a fashion that is invariant under the EDC or he could modify the data and not affect the AICF.) One can avoid this problem by isolating the EDC or AICF in the last granule, adjusting the quanta size or padding

the data if necessary to accomplish this. (An AICF can be used only with a stream cipher mode that exhibits forward error propagation, not with the CFB mode illustrated here.) However, this need to segregate the EDC or AICF imposes a throughput penalty and may introduce some complexity when plaintext chains are sub-block size. For example, to encipher 32 bits of data and a 16-bit EDC, the DES must either adopt a 16-bit quanta for enciphering everything or it must change quanta size from 32 bits for the data to 16 bits for the EDC. The first approach is simpler but requires three DES operations per 48-bit data-EDC chain, whereas the second, more complex approach requires only two DES operations. If this lack of error propagation were not a concern, all 48 bits could be enciphered using the output from one DES operation. A CEDC, as described above for autokey mode, also can be used to provide an authenticity and integrity checking capability.

2.3.4 An Application Example: Secure Network-based Distribution of External Software

The monolithic TRM design presented in section 2.2.2 suffers from a dearth of secure T&A storage. In order to distribute external software using this design, the vendor requires a secure communication path between himself and each TRM. Even in system designs where secure T&A storage is available, network-based distribution of external software may be preferred. Secure communication facilities also may be used to transmit accounting or debugging information to a vendor, so these facilities are important in all system designs. The following discussion describes how to provide secure communication using the cryptographic techniques developed in this chapter. This example introduces the security requirements usually associated with connection-oriented communication and presents some common techniques employed to achieve these requirements. Chapters 3 and 4 show how these requirements and techniques are applicable to the problem of computer system design to protect external software.

First it is necessary to define what is meant by secure TRM-vendor communication. Communication between the TRM and the vendor is effected by exchanging messages on a full duplex connection (virtual circuit) using some communication facility, e.g., a public packet switched network [15] or the dialup phone network. Assume that some standard transport-level communication protocol [25] is employed, providing a connection that is reliable in the face of (non-malicious) errors. The security requirements for this application have been studied extensively and are readily stated.

1. The text of messages must be concealed.

2. Characteristics of the connections should be hidden, e.g., the length of messages and the identities of the ends of the connection. Observation of characteristics such as these is termed *traffic analysis.*

3. The authenticity and integrity of each message must be guaranteed.

4. Each message must be ordered with respect to other messages transmitted on the connection.

5. The timeliness (currentness) of the connection must be ensured.

To achieve these requirements an additional layer of protocol, a security protocol, is introduced. This protocol lies above the transport layer[7] and below the application protocols used to distribute new releases of external software, to report usage statistics from the TRM, etc. Figure 2-14 illustrates the format of messages in the security protocol. In steady state operation, the security protocol accepts each message generated by an application, prefixes it with a sequence number and a control field and appends an EDC or AICF. The resulting message is encrypted in its entirety and delivered to the transport protocol.

---

[7] A properly designed transport layer protocol can provide the facilities required for secure communication with the addition of encryption. However most existing transport protocols do not provide these facilities and thus a separate protocol layer is introduced here.

Figure 2-14: Message Format for Secure Connection Application

To provide concealment and a basis for authenticity and integrity verification, the entire message is encrypted using a block chaining technique such as PCBC or CBC mode. (The control field can be used to indicate if padding was needed and, if so, how many padding characters were inserted.) These modes are simple, convenient and well suited to this application. The sequence number is large enough, say 32-bits, so that it does not cycle during a connection. To prevent duplicate sequence numbers from being generated by the ends of the connection, the sequence number space is divided in half and each end numbers messages using its half of the space. For example, one end could count using odd sequence numbers and the other end could use even sequence numbers. By placing the sequence number at the head of the plaintext chain it serves as an in-block IV. The sequence number also orders all messages on a connection, fulfilling the fourth requirement. The EDC or AICF at the end of the message is checked to determine the authenticity and integrity of each message in accordance with the third requirement.

The second requirement, preventing traffic analysis, can be met in part by padding messages and transmitting *dummy* messages to hide length and frequency of transmission characteristics. However, this technique wastes communications bandwidth and may be too expensive to be feasible. Concealing origin/destination patterns is even harder and cannot be accomplished on an *end-to-end* basis in most communication networks. Through origin/destination analysis an attacker could

learn the identities of clients of various vendors, and by examining the volume of text transmitted he could learn which programs were being distributed. Some vendors may be concerned about these threats posed by traffic analysis and will have to institute appropriate countermeasures (see [16]) but in most cases vendors will probably ignore such threats.

The final requirement calls for appropriate key distribution techniques and a connection initiation procedure utilizing a **challenge-response** protocol. To illustrate these measures consider the following scenario for a secure connection between a TRM and a vendor. Key distribution in this application is quite simple. (For more complex key distribution environments, one might use a public-key cipher in ECB mode to distribute a DES session key, as described in Chapter 5.) At the time of manufacture, or thereabouts, a secret *master key* is generated and loaded into each TRM by the vendor. This master key is different for each TRM and is known only to the vendor. To enable secure communication, the TRM establishes a connection to a vendor computer using the transport protocol. (The assumption here is that the TRM initiates the connection since the vendor is expected to be available via a network at all times, but the TRM may be attached to a network only when required.)

The TRM identifies itself to the vendor by transmitting its (unique) serial number unencrypted. The vendor uses that serial number to lookup the master key for the TRM and generates a random *session key*, to be used only for this connection. The vendor then enciphers the session key under the TRM master key and transmits it to the TRM where it is deciphered and used for further secure communication. The use of a distinct session key for each connection offers several advantages since the same plaintext enciphered under different keys yields different ciphertext. Thus, the IVs used here need be unique only on a per-connection basis to provide adequate concealment. Also, messages from previous connections between the

vendor and this client or connections between the vendor and other clients cannot be replayed or misrouted to confuse either end of the connection (the AICF or EDC would almost certainly be invalid when enciphered under a different key).

With a session key in place, the vendor and the TRM are in a position to challenge one another to verify the time integrity of the connection. Since the vendor generated the session key, he knows the connection is current if the TRM can send messages that pass the usual integrity and authenticity checks (since the messages are enciphered under the session key). Thus there is no explicit challenge carried out by the vendor. However, the TRM, must carry out a challenge protocol to establish that the session key just received is current. The TRM effects this challenge by generating a random bit pattern, encrypting it using the session key and transmitting it to the vendor. The vendor decrypts the bit pattern, transforms it in some predetermined fashion, e.g., complementing half of the bits in the pattern, encrypts this response to the challenge and transmits it to the TRM. The TRM decrypts and checks this response and if it is correct, the timeliness of the connection is verified. This prevents either end from being tricked by a recording of a prior connection initiation sequence. Once this procedure is completed, regular message transmission can begin. (The messages exchanged during secure connection initiation are distinguished from later traffic through appropriate values in the message control field.)

2.3.5 Parameters for Actual Ciphers

To complete this discussion of cryptographic techniques, it is necessary to project appropriate values for cipher parameters, based on existing ciphers and implementations, just as processor capabilities were projected in section 2.1.2. The DES serves as our paradigm for conventional ciphers since it is the most thoroughly studied, modern conventional cipher described in the open literature and since there

are a number of hardware realizations on which projections can be based. The DES operates on 64-bit blocks of text and it employs a 56-bit key. The algorithm performs an initial permutation on the input block and divides it into two 32-bit half-blocks. A *round* of the cipher involves expanding the half-block, adding in selected key bits, performing a substitution and a permutation and then adding in the other half-block and exchanging the half-blocks. Sixteen of these rounds are performed and the half-blocks are concatenated and permuted again to complete the encryption/decryption process.

The fastest current DES implementation (a 4-chip set developed by Fairchild) transforms a 64-bit block in about 3.2μs and requires another 1.6μs to load or unload the data (8 bits at a time), for maximum throughput of about 13 Mbits/s [14]. This chip set, like many other implementations, allows loading of input while the algorithm is executing. Discussions with the designer of this DES chip-set indicate that much faster, single-chip implementations could be produced over the next 3-5 years if suitable demand develops. The projected implementations will be capable of transforming a 64-bit block in 500-1000ns, corresponding to a bandwidth of 64-128 Mbits/s. (The data paths for loading and unloading are likely to be 16 or 32 bits wide for the intended applications.) Even if the next generation of DES chips do not quite achieve this speed, many of the protection mechanisms proposed in this thesis, most notably encrypted storage designs in which primary memory is packaged with the processor, can be implemented without significant performance problems.

The algorithm developed by Rivest *et al.* (the RSA algorithm) serves as the paradigm for public-key ciphers for several reasons. The RSA algorithm is the most widely known and carefully studied public-key cipher, one for which a hardware prototype has been constructed and tested, and the only public-key cipher that supports the double transformation technique for authenticity and integrity

verification described in section 2.3.1. The algorithm encrypts and decrypts blocks of data by exponentiation with respect to a modulus that is the product of two large primes. The encryption and decryption keys are the exponents. Since this algorithm is not a standard no specific block size has been mandated, but to provide security comparable to that of the DES, blocks (and keys) should be about 320 bits in length [17]. (Public-key cipher block and key sizes are generally much larger than those for conventional ciphers because an attacker can carry out only an exhaustive search for a conventional cipher key, but he can search for a secret PKC key using the mathematical structure of the public-key cipher.) This block size could be changed to better fit application requirements, however decreasing the size weakens the cipher and increasing it reduces the encryption/decryption rate. As noted earlier, the prototype RSA single-chip implementation exhibits a projected throughput of about 5 Kbits/s [28].

2.4 Conclusions

The first portion of this chapter described in greater detail the computer system model used throughout the remainder of this thesis. This model describes a fairly conventional, bus-oriented 32-bit computer that is characteristic of many current mini- and microprocessor designs. The model details introduced in this chapter are those required to design the protection mechanisms developed in Chapters 3 and 4. However, not all of the protection mechanisms depend on all of the system characteristics described here. Thus, some of the protection mechanisms are independent of many system details.

The second portion of this chapter examined tamper-resistant modules (TRM) in detail and described how external software could be protected in a computer system based on a monolithic TRM design. The TRM concept is important since it

embodies all of the physical protection characteristics that depend on the level of security required in a particular environment. In this fashion none of the other protection mechanisms developed throughout the thesis need deal with physical protection issues. The monolithic TRM design presented in this chapter might be adequate for some applications but it exhibits a number of limitations, e.g., it cannot support demountable storage media. This motivates the use of cryptographic techniques to overcome these limitations. The last portion of the chapter introduced some terminology, concepts and techniques from modern cryptography. This material is included to provide background for readers who are not familiar with this area. The explanations provided here are not intended as a general primer on cryptography, but rather are directed toward the specific application areas encountered in the thesis.

Chapter Three

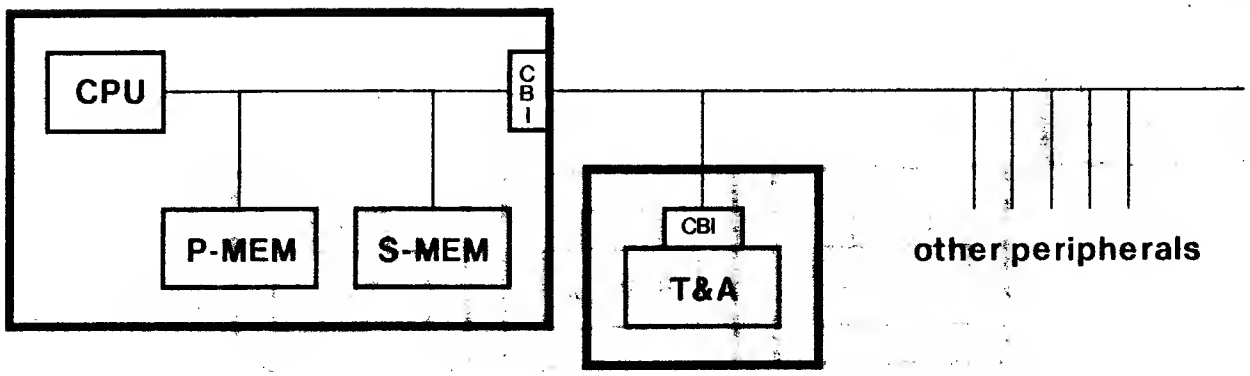
An Encrypted Bus Approach to Protecting External Software

The arsenal of cryptographic techniques presented in section 2.3 suggests several ways to protect external software in computer systems without enclosing all of the security relevant components in a single TRM. This chapter explores in detail an approach based on viewing a computer system as a miniature communication network. In this approach, each security relevant component (or collection of components) is enclosed in a TRM and communicates with other components over a physically unprotected bus. Each TRM is equipped with a special *cryptographic bus interface* (CBI) that provides secure communication among the TRMs. The major advantage of this approach over the monolithic TRM design is that it permits distribution of the secure components among several TRMs. Thus it becomes possible to incrementally change a system through selective replacement or addition of TRM-packaged components (for maintenance or expansion) and many problems associated with TRM sizing become more manageable. One might even provide a form of demountable storage in this type of system, by packaging the media and its access hardware in a demountable TRM, although such storage would not be competitive with conventional, demountable media in terms of cost or convenience.

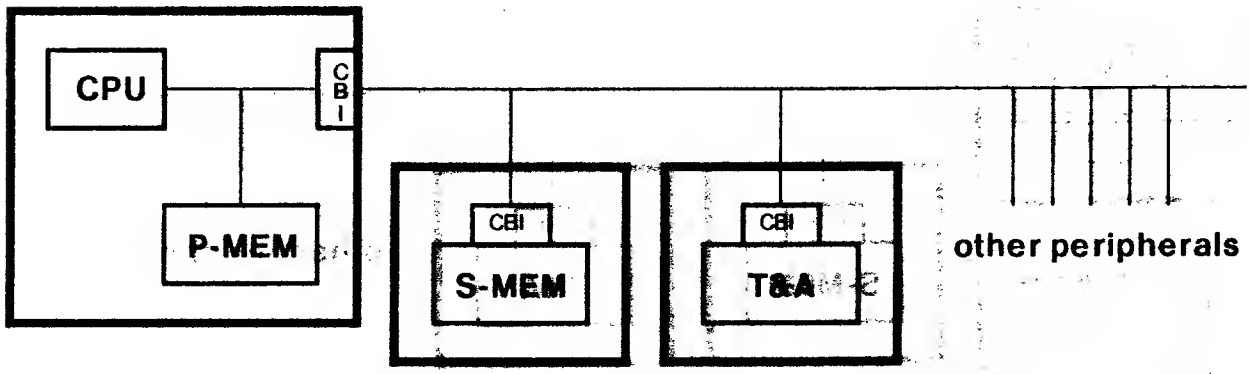
3.1 Configurations and Overview

The system configurations pictured in Figures 3-1 and 3-2 characterize the ways in which TRM packaging can be employed in this communication security design approach. SYSTEM A represents the smallest change from the monolithic TRM design, providing a separate TRM only for the transfer and archival (T&A) storage device. SYSTEM B provides greater flexibility by employing separate TRMs for secondary as well as T&A storage devices. In both of these configurations the organization of the processor and primary memory, i.e., the presence or absence of cache or a dedicated memory bus, is irrelevant since they are contained wholly within a single TRM. In these configurations, the cryptographic bus interface (CBI) for the main TRM (the TRM containing the processor) also operates like the secure bus coupler (SBC) described in section 2.2.2, i.e., it keeps unencrypted traffic in the main TRM from appearing on the bus outside this TRM and it restricts access to primary memory locations by DMA devices outside the main TRM. In SYSTEM C and SYSTEM D the maximum degree of flexibility available in this design approach is attained as each device is packaged in a separate TRM. Here the choice between single and dual bus configurations has a significant impact on the design, as detailed in the following sections.

The techniques described in section 2.3.4 could be applied directly to this design, but the characteristics of bus communication differ enough from those usually encountered in general communication environments to warrant modifying those techniques. For example, since bus operations involve very few bits (about 32 bits of data or address plus some control bits), the additional information required for security (e.g., EDCs and sequence numbers) represents a significant percentage increase in the amount of data transmitted. Transporting this extra information requires either additional bus lines, increasing the cost of bus interfaces, or additional bus cycles, increasing transaction time and reducing bus availability. In a

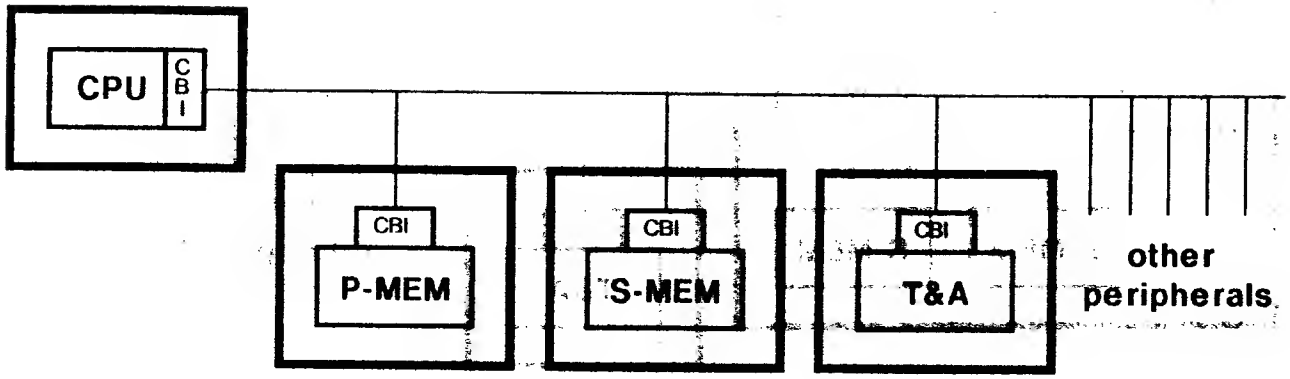


SYSTEM A

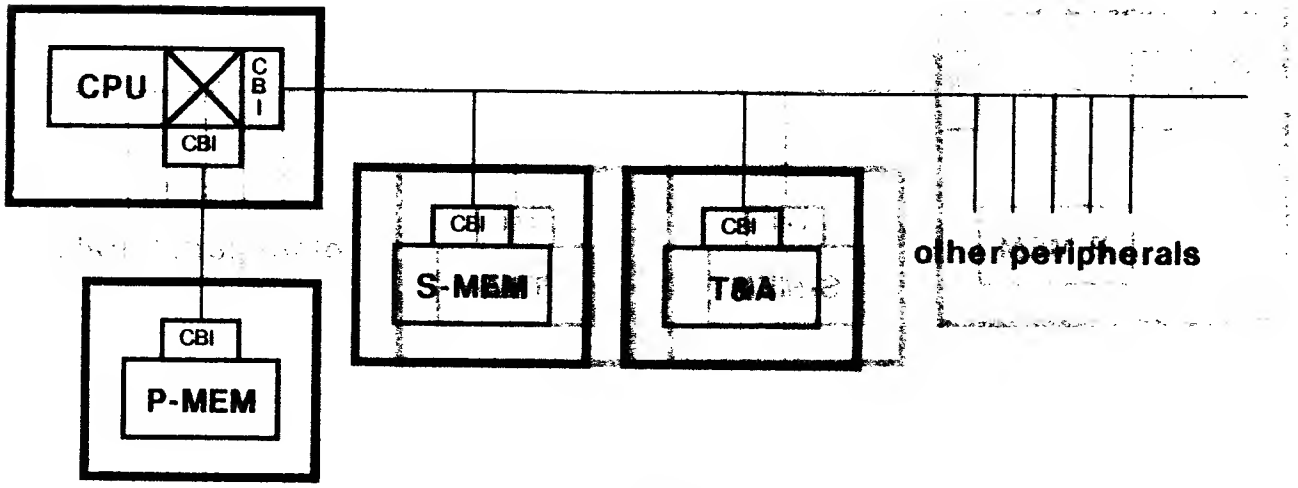


SYSTEM B

Figure 3-1: Two System Configurations Employing TRMs with CBIs



SYSTEM C



SYSTEM D

Figure 3-2: Two More System Configurations Employing TRMs with CBIs

similar vein, the high speed, low delay nature of bus transmission means that any bandwidth limitations and delays introduced by cryptographic and protocol techniques could dramatically slow down the system. Thus, in adapting communication security measures to the bus environment, special care must be taken to minimize delays, maximize bandwidth and reduce the amount of additional information transported with each operation. Moreover, the additional hardware required for secure bus communication must not significantly increase the cost of the computer system.

The cryptographic techniques developed in this chapter are carefully tailored to the bus environment, taking advantage of the highly structured nature of transactions and the high reliability of bus communication to minimize overhead on bus transactions. Special cipher modes and error detection techniques are employed to minimize the number of additional bits transmitted and the delay associated with securing bus transactions. In engineering protection mechanisms for the encrypted bus approach, three classes of transactions involving TRM-packaged system components are distinguished:

1. Processor-generated references to primary memory

2. Transfers between primary memory and DMA peripherals

3. Transactions used by the processor to control DMA peripherals and used by these peripherals to interrupt the processor

The first and third transaction types are referred to as *simple* in contrast to the *aggregate* transactions used to effect DMA transfers. Transactions of the first type constitute the bulk of bus traffic. Any reduction in bandwidth or increase in delay experienced by these transactions significantly affects system performance. Transactions used for DMA transfers constitute a much smaller percentage of all bus traffic and they are qualitatively different in that they deal with *aggregates* of

data. This latter characteristic makes it possible to reduce per-transaction overhead by validating a data aggregate as a whole rather than checking each word of the aggregate as it is transferred. The last type of transactions, those employed in the control of DMA[8] peripherals, are very infrequent compared to the other types of transactions, and thus system performance is affected only slightly if these transactions become somewhat "slower."

3.2 Security Requirements for the Encrypted Bus Approach

As noted in Chapter 1, vendors have two major requirements for protecting external software in this context: preventing release of and detecting modification of information. In computer systems based on the encrypted bus approach, the bus constitutes the only vulnerable, security relevant portion of the system and thus bus transactions are the principal target for an intruder. Even though the bus is a broadcast transmission medium, the flow of data among devices is actually connection-like in nature, not broadcast oriented. The flow of data among TRM-packaged devices corresponds to the three types of transactions described in the preceding section, i.e., data flows between the processor and primary memory, between primary memory and DMA peripherals and between the processor and these peripherals. The data flow is thus implicitly segregated into distinct (duplex) connections, one between each pair of devices as described above. Hence the requirements for secure bus operation are, at a high level, the same as those for general purpose, connection-oriented communication environments as described in section 2.3.4: preventing disclosure of message text and traffic analysis, ensuring message authenticity, integrity and ordering, and ensuring the timeliness of the connection.

---

[8] All of the TRM-packaged peripherals are assumed to be DMA devices. If non-DMA peripherals were employed, this same class of transactions would be used for control purposes.

These requirements are easily translated to the context of bus communication among TRM-packaged devices. Here, disclosure of message text refers to exposure of the data in **PRESENT-DATA** operations. Traffic analysis in this context involves exposure of the addresses in **PRESENT-ADDRESS** operations, identification of the TRMs engaged in each transaction, determination of operation types and observation of patterns of data transfer. The authenticity, integrity and ordering requirements are directly applied to the bit patterns representing each operation on the bus. Thus each received bit pattern must be checked to verify that it was generated by a CBI-equipped TRM in the system, that it was not modified en route and that it arrived in proper order with respect to other operations between this device and its partner in this transaction. The CBIs must be initialized to a known state and must verify the timeliness of connections before data transmission may begin.

In this context traffic analysis may be a more serious threat than in the client-vendor communication scenario described in section 2.3.4. For example, by observing the pattern of references to memory made by a processor, noting the locations accessed and whether the processor reads or writes these locations, an attacker may be able to deduce quite a bit about the nature of the procedure being executed. Similar observations of data transfers between primary memory and cache or between secondary or T&A storage and primary memory provide clues as to the nature of the procedure. How much information can be gained in this fashion depends to a great extent on the system configuration. For example, **SYSTEM C** and **SYSTEM D** provide more opportunities for traffic analysis than **SYSTEM B** which in turn provides more opportunities than **SYSTEM A**. Note that adding a cache to the processor in **SYSTEM C** or **SYSTEM D** reduces the opportunities for traffic analysis since most references to primary memory are satisfied by the cache and thus do not result in transactions outside the processor TRM.

The amount of information gained through traffic analysis also depends on the extent to which characteristics of traffic are visible. In the worst case an attacker can discern the addresses in **PRESENT-ADDRESS** operations, as well as identify the operation types. In a less severe scenario an attacker could identify the TRMs involved in a transaction and determine the transaction type but would not be able to discover the specific locations involved in the transfer. Although it would be preferable if all traffic analysis were prevented, as in the monolithic TRM design, this is prohibitively costly to achieve because of bus characteristics and so some compromise is required.

While it is possible and practical to conceal the addresses in **PRESENT-ADDRESS** operations, it is not feasible to hide origin-destination patterns at the TRM level. An intruder can passively wiretap the bus between each TRM and discover which TRM is transmitting, but not which is receiving. However, bus transactions follow a very simple pattern of a request operation followed by a response, so the intruder can easily determine which TRMs are involved in a transaction. Since the identity of the TRMs involved in a transaction cannot be concealed, the only way to obscure origin-destination patterns is for TRMs to generate dummy transactions at random intervals. Yet if the dummy transactions interfere with genuine bus traffic severe performance degradation may result.

If buses were multiplexed in a time division fashion, with each TRM assigned a time slot to carry out a transaction, the dummy transaction technique could be employed. But the demand access nature of buses and the arbitration schemes commonly employed make this technique infeasible for two reasons. First, a device cannot know in advance whether a dummy transaction would conflict with a genuine transaction in bidding for the bus during an arbitration procedure. Second, even if a dummy transaction were initiated only when there were no genuine demands, the bus would be busy for an interval during which a genuine demand

might arise. In a system with a priority structure for bus arbitration the first problem could be overcome by having the lowest priority device (usually the processor) be the only generator of dummy transactions, but the second problem would remain. Only on a dedicated memory bus with transaction interleaving could the processor/bus coupler inject dummy transactions without degrading bus performance.

The preceding analysis suggests that preventing origin-destination analysis and masking general patterns of bus traffic at the TRM level is infeasible except in limited contexts. Hiding the types of transactions, i.e., preventing an intruder from distinguishing among **read**, **write** and **interrupt** transactions or their extended counterparts, also is infeasible in most contexts because the patterns of bus utilization and/or the duration of the transaction are different for each type of transaction. Thus signals on bus control lines, i.e., lines other than **A/D0-31**, need not be concealed. Only in the context of a dedicated memory bus with transaction interleaving could these transaction characteristics be hidden. (This type of bus arrangement is highly analogous to a simple, full duplex communication link and thus is amenable to link encryption techniques, unlike a general purpose or I/O bus.) Thus, if traffic analysis threats such as these are a major concern, configurations such as **SYSTEM A** or **SYSTEM B** should be considered.

3.3 Securing *Simple* Transactions

This section develops techniques for securing *simple* transactions. These are the transactions used in the control peripherals in all four system configurations and in processor-memory transfers in **SYSTEM C** and **SYSTEM D**. The same protection mechanisms are applied to both types of transactions. Processor-memory transactions will be processed more quickly than control transactions only because

the CBIs at the bus coupler and primary memory will incorporate multiple cryptographic devices and extra bus lines to achieve greater parallelism. (The need to employ additional bus lines to transport error detection information for this type of transaction strongly motivates adoption of the dual bus configuration, **SYSTEM D**, to minimize the cost of the extra lines.) Otherwise, the two transaction types are treated identically, simplifying CBI system design.

For *simple* transactions, encryption is required both for secrecy and to enforce authenticity, integrity and ordering requirements for transactions. Of course the data in **PRESENT-DATA** operations must be concealed, and if traffic analysis is a concern, the addresses in **PRESENT-ADDRESS** operations also must be concealed. In the case of a **simple read** transaction, the bus master must verify that the data returned in a **PRESENT-DATA** is from the location specified in the immediately preceding **PRESENT-ADDRESS**, that the returned data has not been modified in transmission and that it is timely (not a copy of data from a previous bus operation). In the case of a **simple write** transaction the slave must verify the authenticity, integrity and ordering of each **PRESENT-ADDRESS** and **PRESENT-DATA** and the master must do the same for each **ACKNOWLEDGE**. On an **interrupt**, the master must verify the authenticity, integrity and ordering of the vector in the **PRESENT-DATA** and the slave must do the same for the **ACKNOWLEDGE** it receives.

Note that the ordering requirements set forth here are strictly per-connection, i.e., transactions between the processor and primary memory are explicitly ordered among themselves but are not explicitly ordered with respect to transfers between DMA devices and primary memory. Thus the requirements explicitly impose local ordering (on each connection) but not an explicit global ordering. Yet global ordering is important. For example, data may be written into primary memory by the processor and then transferred to secondary storage. These two transfers take

place over two distinct connections and thus do not fall under the explicit, per-connection ordering requirements set forth above. However, the processor initiates all data transfers, either directly or through control of DMA device activities, and thus it imposes a global ordering of these transfers even though the transactions are not explicitly, globally ordered. For example, in the example noted above, the processor will not initiate the DMA transfer to secondary storage until it has written the data into primary memory. Thus global ordering is imposed implicitly by the processor, relying on explicit, per-connection ordering of transactions.

Readers who do not wish to delve into the details of how simple transactions are secured should skip to section 3.4 (page 132), to the discussion of how aggregate transactions are secured.

3.3.1 Securing *simple read* Transactions

The security requirements stated above for a **simple read** constitute a relaxation of those stated in section 3.2 in that the slave does not carry out any authenticity, integrity or ordering checks on a **PRESENT-ADDRESS**. These relaxed requirements allow an intruder to submit a spurious **PRESENT-ADDRESS** to the slave and receive an encrypted **PRESENT-DATA** response. A spurious **PRESENT-ADDRESS** will not adversely affect system security so long as the resulting **PRESENT-DATA** cannot be used to spoof the master, i.e., the master must be able to verify that a **PRESENT-DATA** is an authentic response to the **PRESENT-ADDRESS** just issued by the master. (Of course, the concealment mechanisms also must not be affected by this relaxation.) If the checks noted above are carried out on each **PRESENT-DATA**, then the master cannot be spoofed in this fashion. Thus this relaxation of requirements does not introduce any new vulnerabilities and it avoids the adverse performance effects associated with calculating and transmitting an error detection code as part of each **PRESENT-ADDRESS** in a **simple secure read.**

For processor-memory transactions, the cryptographic facilities must exhibit a throughput sufficient to keep pace with bus operation and must introduce minimal delay. Since both addresses and data are to be concealed on the bus, cryptographic devices must exhibit a bandwidth of about 106-213 Mbits/s at peak bus utilization. (These figures are for a cacheless processor; for a cache-equipped processor an even higher bandwidth is required.) Since the maximum bandwidth projected for single-chip DES devices ranges from about 64-128 Mbits/s, these devices are not capable of meeting peak bus traffic requirements in comparably scaled systems. (The assumption here is that one will employ fast DES chips in conjunction with a fast bus and fast primary memory, and slow DES chips with a slow bus and memory.) Moreover, these DES devices require about .5-1.0μs to transform a 64-bit block, a processing delay equivalent to five bus cycles, and this delay may be a serious problem even if the bandwidth is adequate. In **SYSTEM C** and **SYSTEM D** the memory and the bus coupler CBIs must keep up with processor-memory transactions and this is a difficult task.

A stream cipher mode of operation, rather than a block mode, is essential here because of the need to maximize throughput and to minimize delay. Only about 32-bits are encrypted in each bus operation, creating an immediate granularity mismatch between the plaintext and a block mode of operation. A block mode also imposes a delay (T_c) to encrypt and decrypt the data since the algorithm cannot be executed until the text is available. To better understand why block mode was rejected, consider the processing steps involved in a **simple secure read** based on ECB mode encryption. The event graph and timing diagram in Figure 3-3 illustrates these steps. The address in the **PRESENT-ADDRESS** and a unique IV/AICF are encrypted using ECB mode (C1), transmitted (T1,T2) and deciphered at primary memory (C2). (The IV/AICF used here is just a sequence number.) The data is retrieved (A), enciphered along with the incremented IV/AICF (C3), and transmitted to the processor (T3,T4) where it is deciphered (C4) and the AICF is checked (=).

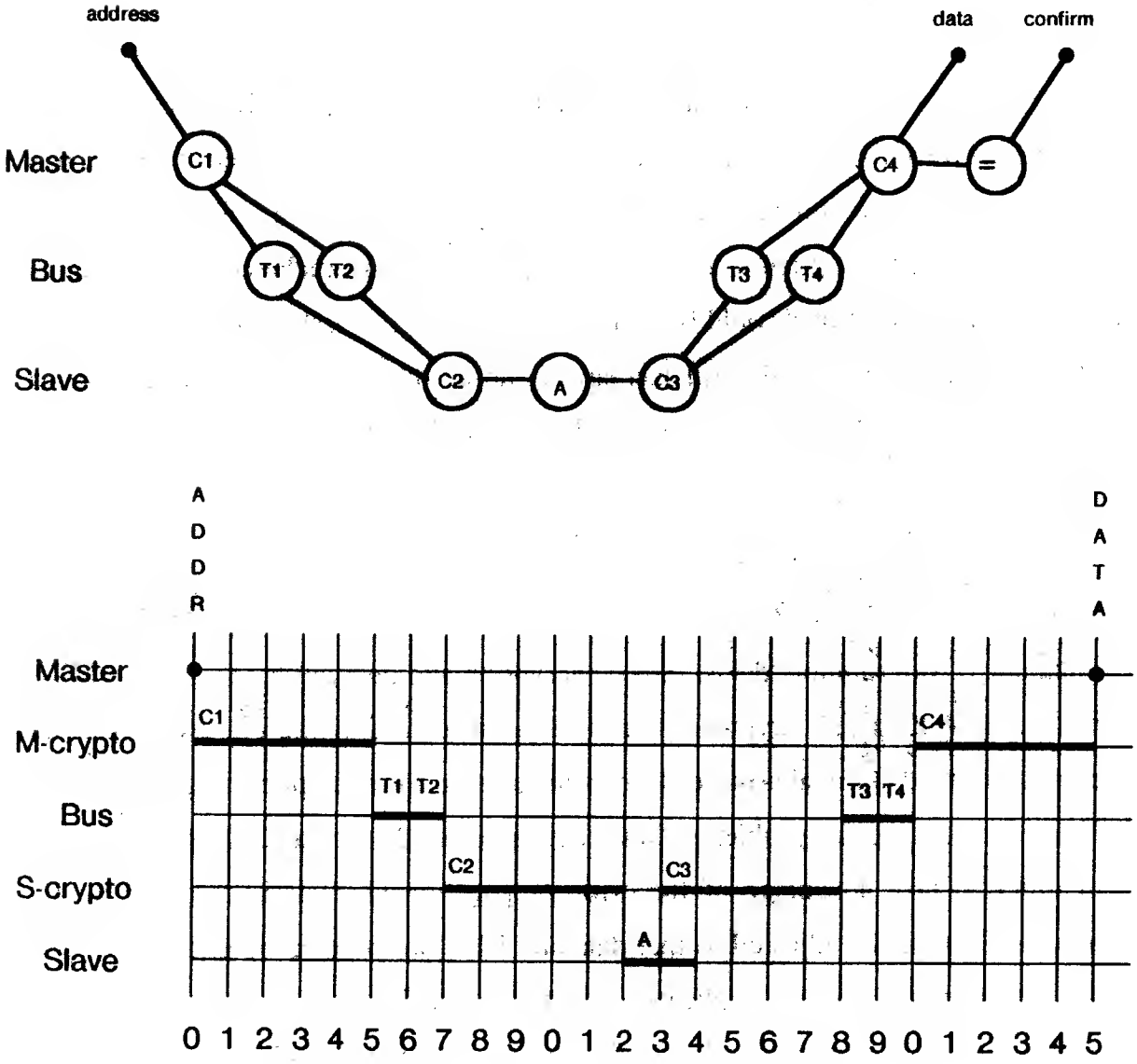


Figure 3-3: Event Graph and Timing Diagram for an ECB Mode Secure Read

The total transaction time for this *ECB mode* **simple secure read** is $4T_c + 4T_t + T_a$ (25 bus cycles), as compared to a **standard read** time of three bus cycles. The timing expression is easy to derive since there are no parallel processing steps in the

event graph, and that is the root of the performance problem. To support the maximum transaction rate as a standard system, one would have to employ additional cryptographic units, interleave transactions and add another 32 bus lines (since twice as many bits are transmitted here as in a **standard read**). These changes would significantly increase the cost of CBIs. Even with these added facilities, this design exhibits an inherent delay that translates into over a 730% increase in effective memory access time for a cacheless processor. For cache-equipped systems a **standard extended read** could be secured in an analogous fashion, but the effective memory access time would still increase by about 48-120%. These delays are so great as to preclude the use of this mode even with the CBI enhancements noted above.

A stream cipher mode of operation provides opportunities for parallelism and for precomputation of the crypto bit stream, so that a high throughput rate can be maintained with minimal delay. Since encryption and decryption are accomplished by adding (modulo 2) cryptographic bit stream to text, if the bit stream can be computed in advance, almost no delay is introduced for encryption and decryption. However there are two problems if a stream cipher mode such as CFB is used. First, in order to take advantage of the error propagation characteristics of CFB, the quanta size must be adjusted so that data and EDC are covered by different crypto bit stream quanta. In this application the data is usually 32-bit words or addresses, so the quanta size would probably be 32 bits. This quanta size halves the bandwidth provided by the cipher, a serious problem given the timing of DES calculations and bus cycle times for the systems of interest. Second, there is a delay (T_c) in providing the crypto bit stream for the EDC, since this bit stream cannot be generated until the data being protected has been encrypted. (Remember, the ciphertext must be fed back into the algorithm to generate the next quanta of crypto bit stream.)

To avoid these problems of reduced bandwidth from cryptographic devices and substantial delays for transmission and checking of EDCs, a degenerate form of autokey cipher mode is used. This stream mode employs no feedback from cleartext, ciphertext or the crypto bit stream. Instead, each block of crypto bit stream is generated using a unique IV. Each IV is formed by concatenating a *bit stream ID* and a counter that is incremented each time the algorithm is executed. The bit stream ID distinguishes cryptographic bit streams generated under the same key. This stream cipher mode exhibits several very important properties. For example, *n* cryptographic devices can be used in parallel to generate a single bit stream by initializing the counters to the values *1* through *n* and incrementing by *n* each time (using the same bit stream ID for all). This makes the output appear as though it came from a single, fast cryptographic device and allows using different crypto device configurations at each end of a connection, e.g, units of differing speeds or different numbers of units to generate the same bit stream. Moreover, since no feedback is employed, crypto bit stream blocks can be generated at the maximum rate for crypto devices that allow loading the next input while the algorithm is being executed (a common design feature in many DES chips).

For securing bus transactions, each TRM generates two distinct bit streams for each device with which it communicates: a *transmission bit stream* and a *reception bit stream.* Thus, for each connection, one crypto bit stream is used to encipher bus operations transmitted by the TRM and another bit stream is used to decipher bus operations that the TRM receives. (Of course these terms are relative since a transmission bit stream at one TRM is a reception bit stream at the TRM that is the target of the bus operation.) In communications parlance a different crypto bit stream is associated with each independent simplex channel. The endpoints of each connection generate the two bit streams for that connection in synchrony so that IVs need not be transmitted and so that the receiver of an operation can precompute the

bit stream needed to decipher the incoming operation. The use of a different bit stream for each channel is important. If the same bit stream were employed for more than one simplex channel, it would be necessary to impose additional constraints to prevent two TRMs from transmitting data enciphered under the same bit stream.

This stream cipher mode permits encryption and decryption of bus operations with almost no delay, assuming a sufficient number of cryptographic chips are employed in parallel. However, this stream mode does not provide any error propagation for authenticity and integrity checks and thus a cryptographic error detection code (CEDC) must be employed for that purpose. Using a CEDC, the generation of crypto bit stream for encrypting and decrypting data is independent of the CEDC calculation. Thus one DES chip can be dedicated to calculating the CEDC and crypto bit stream generation can proceed in parallel using other DES chips.

Since stream mode encryption and decryption can take place with no appreciable delay and can keep pace with any transmission rate (using multiple units in parallel), the transaction time for a **simple secure read** based on this design exceeds the time for a **standard read** only by the amount of time devoted to the CEDC generation and checking. The simplest way to calculate a CEDC in this application is to encrypt the data to be protected using ECB mode, and to transmit a portion of the resulting ciphertext block. (It is not necessary to transmit the entire CEDC block since the receiver of the data can perform the same calculation on the data and compare the appropriate portion of the result to the received CEDC bits.) If a full, 16-round DES encryption is performed to calculate the CEDC, the delay introduced by this operation is T_c, no better than the delay provided by CFB mode. However this delay can be reduced by operating on the plaintext for less than the full 16 rounds and by transmitting a portion of the result encrypted using stream mode.

The idea is to reduce the time required for CEDC calculation but to maintain security by using enough rounds and by stream encrypting the resulting CEDC. After five rounds of the DES, every bit of the output is a complex, non-linear function of every bit of the input and of every bit of the key. The error propagation provided by five rounds of the DES makes it impossible to change data in a fashion that is invariant under this CEDC. Also, if the CEDC is stream mode encrypted before transmission, the intruder cannot discover the value of a CEDC except through cryptanalysis of the full 16-round DES. In order to tamper with data covered by the CEDC (and not be detected), the intruder must either be able to predict the CEDC generated on a known input or be able to predict the changes in a CEDC resulting from complementing a bit in a known or unknown input. Because all of the key bits are involved in determining the value of each output bit, each of these tasks is probably equivalent to *breaking* a five-round DES, i.e., discovering the key. As there is no indication that a five round DES can be broken by other than a brute force attack, and since the matching ciphertext required for such an attack is itself encrypted under a full strength DES, there is good reason to believe than an intruder cannot subvert this CEDC scheme.

Figure 3-4 illustrates the steps involved in a **simple secure read** employing the stream mode enciphering/deciphering and the CEDC scheme described above. The master begins by generating its transmission crypto bit stream (C1) using the stream cipher procedure described above. The address in a **PRESENT-ADDRESS** is enciphered using 32 bits of that bit stream (X1) and the result is transmitted (T1). The address is deciphered at the slave (X2) using the corresponding portion of the slave reception bit stream (C2). The address is used to retrieve a word from memory (A). The slave generates its transmission crypto bit stream (C3), enciphers the retrieved data (X3) using 32 bits of this bit stream and transmits the result in a **PRESENT-DATA** (T2). The master deciphers this operation (X4) using the corresponding portion of its reception bit stream (C4).

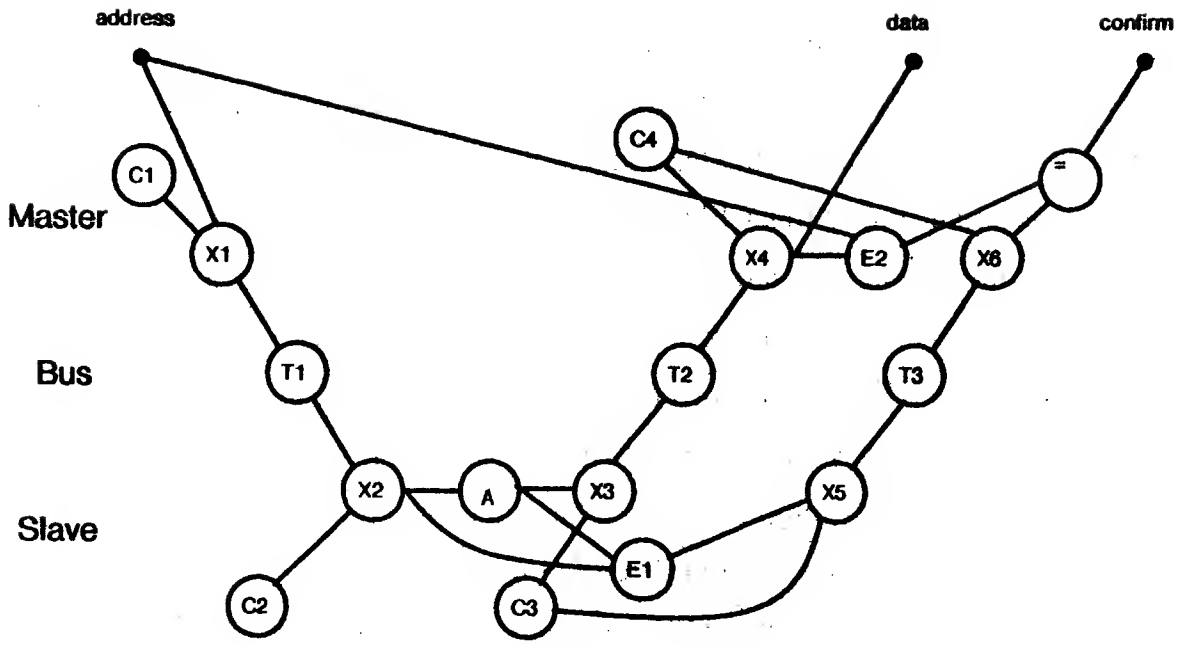


Figure 3-4: Event Graph for **a simple secure read**

While steps *X3*, *T2* and *X4* are taking place, the slave can calculate the CEDC (E1), using both the address and data as input. Once the CEDC is available, a portion of it is encrypted (X5), using more of its transmission bit stream from *C3*, and the result is transmitted to the master (T3). At the master, once the data is decrypted (X4) using corresponding master reception bit stream from *C4*, it is concatenated with the address to calculate the CEDC (E2). When the CEDC calculated at the slave arrives and is decrypted (X6), it is compared (=) with the corresponding portion of the CEDC calculated at the master to verify the authenticity, integrity and ordering of the transaction. The decryption of the slave CEDC (X6) and the comparison (=) can be re-ordered and re-associated (the master CEDC can be added to the appropriate crypto bit stream and the result

compared to the incoming, encrypted slave CEDC) if performance is improved by this alternative ordering of steps.

Since master and slave generate different transmission bit streams, neither will transmit data enciphered under the same bit stream that the other is using to encipher data, regardless of attacks, and thus concealment is ensured. If the data in the **PRESENT-DATA** is modified or if the data is not from the requested location, this will be detected since the CEDC is a function of both. The timeliness of the transaction also is assured by the use of different crypto bit stream for each bus operation and by the CEDC. An old transaction will be improperly decrypted because of the uniqueness of the crypto bit stream and this will result in a mismatch in the CEDC check. The intruder cannot compensate for the differences in the crypto bit stream unless he can calculate CEDCs, a feat made impractical by the scheme used here. Thus this design achieves all of the security requirements established for **simple read** transactions at the beginning of this section.

The minimum transaction time for this **simple secure read** is $2T_t + T_a + T_e$ (5 bus cycles) as derived from the timing diagram in Figure 3-5. However, the data is available at the master after $2T_t + T_a$, the same as for a **standard read**. Thus unverified data is available at the master with no additional delay from the beginning of the transaction, but total transaction time increases by 66%. A processor employing pipelining might be able to "backup" if data is discovered to be invalid within two bus cycles after its delivery, but most systems will have to abort and shut down under these circumstances. In many cases, it will not be acceptable to deliver unverified data and the master will incur a 66% increase in effective access time. This is clearly unacceptable for processor-memory transactions. However, in a cache-equipped system, a **secure extended read** can be implemented in a similar fashion and the effective average memory access time for verified data increases by only 4-9% in this case. This increase is small enough to be acceptable in most applications.

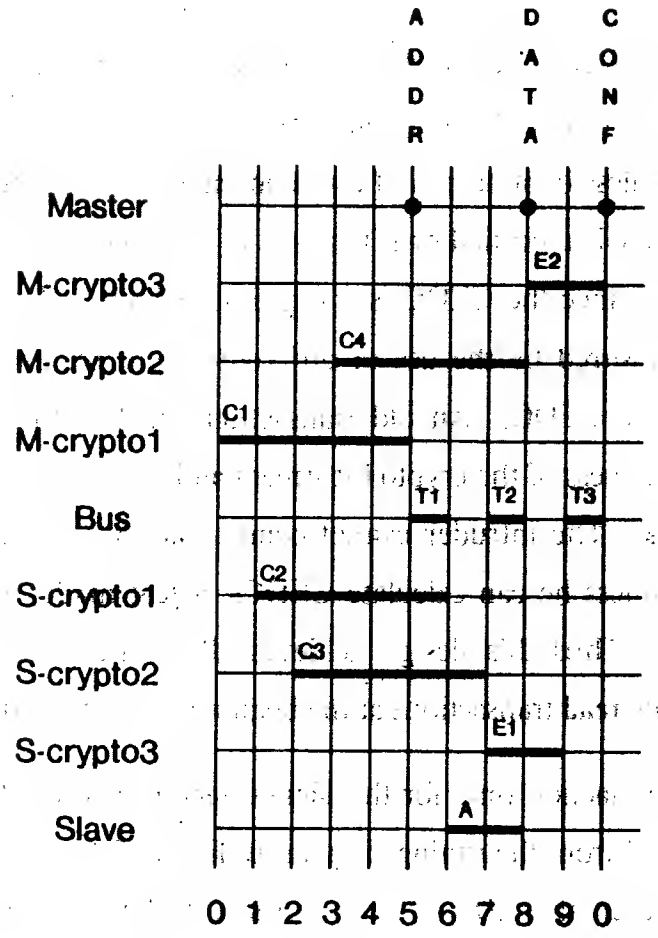


Figure 3-5: Timing Diagram for a **simple secure read**

Delay in delivery of data is not the only concern here. For processor-memory transactions the maximum standard transaction rate should be attainable and bus utilization should not increase significantly. The effective memory access time calculations performed above assume that successive **simple secure read** transactions can be issued at the same maximum rate as **standard read** transactions. Unless the next transaction is allowed to begin before the CEDC of the preceding transaction is transmitted, this maximum rate cannot be achieved. Thus, for processor-memory

transactions, CEDC transmission must be interleaved with address and data transmission. One might attempt to transmit the CEDC on the A/D0-31 lines during the idle cycle in the middle of **simple secure read** and **simple secure write** transactions (see Figures 3-5 and 3-8). However, this idle cycle will not always occur at the time when the CEDC should be transmitted. Moreover, the secure versions of extended transactions do not provide such idle cycles.

This analysis suggests that a separate set of bus lines is required to support interleaving of CEDC transmissions for processor-memory transactions. Sixteen additional lines (CEDC0-15 should suffice for most applications since, if 16 CEDC bits are transmitted for each transaction, an attacker has a 2^{-16} chance of undetectably tampering with a transaction. These bus enhancements (extra lines for CEDC transmission and interleaving of this transmission) are required only for processor-memory transactions, so they affect only **SYSTEM C** and **SYSTEM D**, where the bus segment between the processor and primary memory is unprotected. These enhancements are most easily and economically implemented in a dual bus system configuration, where the existence of only a single bus master makes interleaving feasible and equipment cost is minimized since only two bus interfaces are involved. Thus **SYSTEM D** is strongly preferred over **SYSTEM C**. In **SYSTEM A** and **SYSTEM B** the simple transactions on the exposed bus segment are strictly control transactions and the increased delay due to CEDC transmission on the A/D0-31 lines on this segment should not pose a significant performance problem.

For processor-memory transactions, the CBIs at primary memory and on the memory bus connection to the bus coupler each require four cryptographic devices to maintain the maximum transaction rate. Figure 3-6 shows the utilization of the cryptographic devices, memory and bus lines for a series of six successive **simple secure read** transactions. In each three-cycle transaction, 32 bits of address must be

concealed by the master so a single device (*crypto1*) can supply the needed 64 bits every six cycles. The slave must conceal 32 bits of data and 16 bits of CEDC every transaction, for a total of 48 bits every three cycles (at maximum rate). Two cryptographic devices (*crypto2* and *crypto3*) are used for this task since a single device can generate only 64 bits every five cycles. Finally, one crypto device (*crypto4*) is required to generate the CEDCs, using two bus cycles in each three-cycle transaction to perform five of the 16 rounds of the DES. Since this string of transactions represents a series of processor-memory transactions, the extra bus lines (CEDC0-15) are employed for CEDC transmission.

If the traffic analysis threat is ignored, addresses need not be encrypted and 32 fewer bits would have to be concealed on each transaction. In this case only three crypto units are required at the processor and primary memory, i.e., *crypto1* can be eliminated. Even if addresses in processor-memory transactions are concealed, it is quite likely that address concealment may be omitted for control transactions (those involving the processor and DMA peripherals) since the device register addresses in these transactions provide very little information to an attacker. Unlike processor-memory transactions, the frequency of control transactions is fairly low and there should be enough time between these transactions to allow a single crypto device to precompute crypto bit stream between uses (whether or not addresses are concealed in these transactions). This would free this device for CEDC calculation during these transactions. Thus TRM-packaged peripherals probably require only one crypto unit (changing bit stream IDs as required) for **simple secure read** control transactions.

3.3.2 Securing *simple write* Transactions

The detailed security requirements for **simple write** transactions provide no opportunities for relaxation, unlike **simple read** transactions. The contents of the

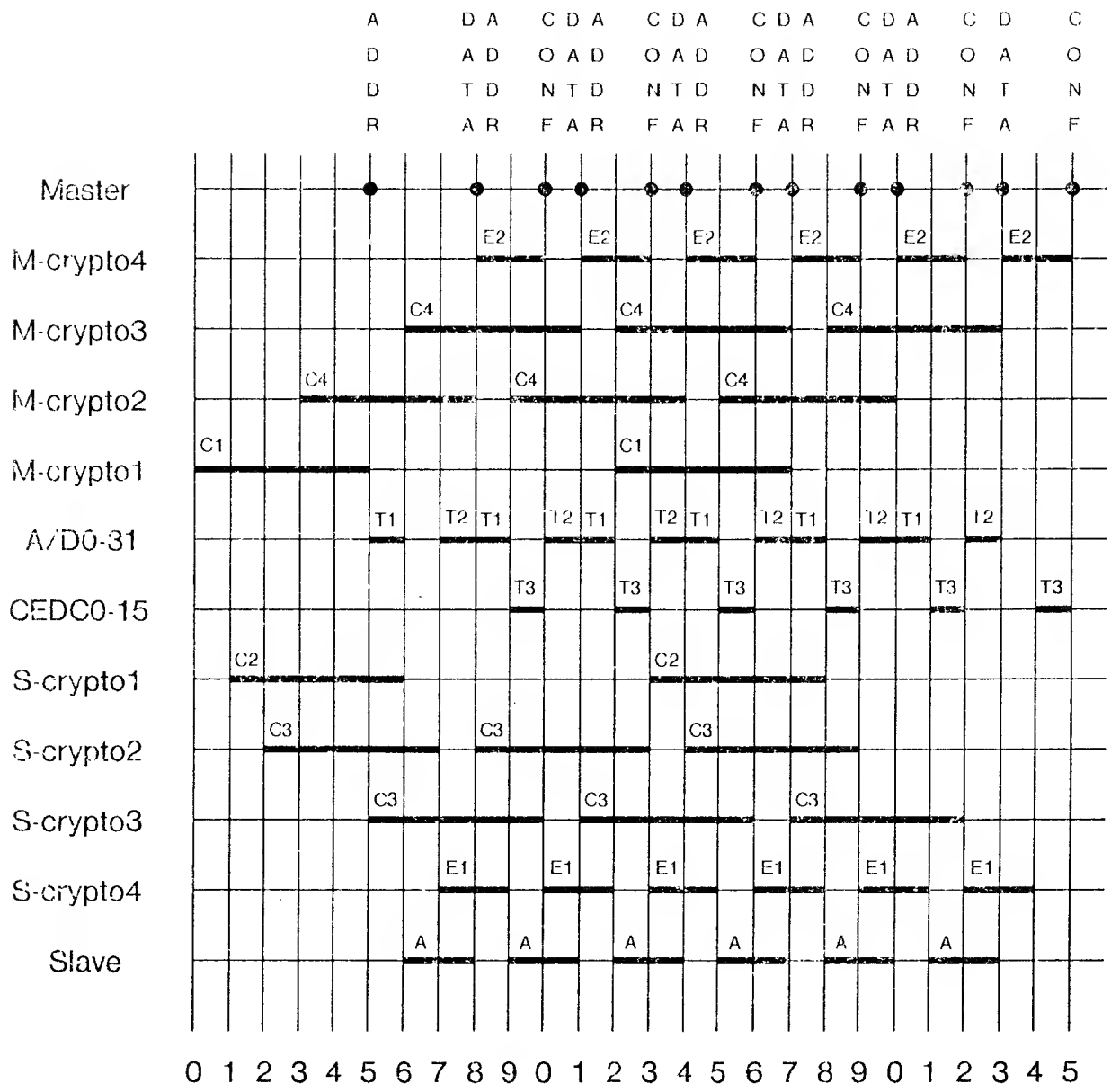


Figure 3-6: Timing Diagram for Successive **simple secure read** Transactions

PRESENT-ADDRESS and PRESENT-DATA must be concealed. The slave must verify that these operations are ordered with respect to other transactions on the connection, and that the address and data are authentic and unmodified. The slave must provide the master with a secure ACKNOWLEDGE verifying the successful completion of the **simple secure write**. These requirements can be achieved using many of the same techniques developed for secure reads. Stream mode encryption and decryption are employed for concealment and the same CEDC technique is applicable here to ensure the authenticity, integrity and ordering for each operation in the transaction. Figure 3-7 shows the event graph for the **simple secure write** resulting from an application of these techniques.

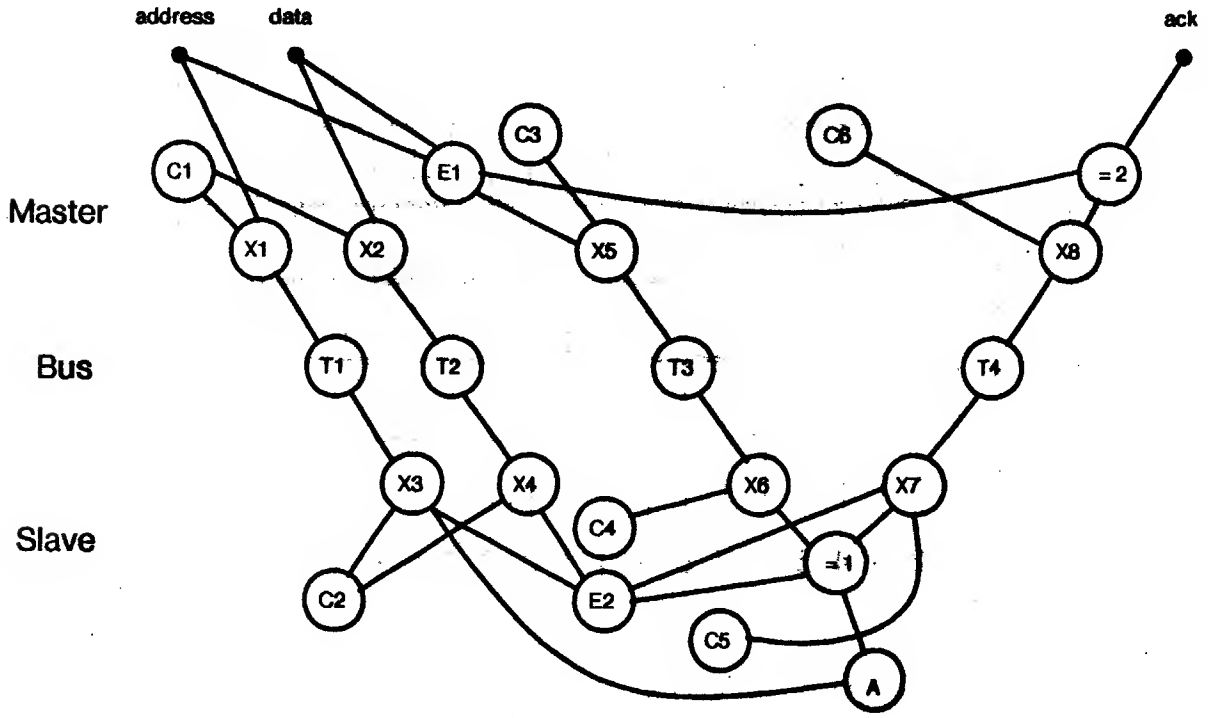


Figure 3-7: Event Graph for a **simple secure write**

The master begins by generating 64 bits of transmission bit stream (C1) for concealing address and data. The address is encrypted (X1) using half of these bits and the result is transmitted (T1) using a **PRESENT-ADDRESS**. The slave receives this encrypted address and decrypts it (X3) using the corresponding portion of the slave reception bit stream generated in *C2*. Back at the master, the data is encrypted (X2) using the remaining 32 bits from the transmission bit stream generated in step *C1*. The result is transmitted (T2) using a **PRESENT-DATA** and deciphered at the slave (X4). At the master, the address and data are used to calculate a 64-bit CEDC (E1), a portion of which (say 16 bits) is encrypted (X5) using a matching amount of additional transmission bit stream generated in *C3*. This CEDC is transmitted to the slave (T3) where it is deciphered (X6) using the corresponding reception bit stream generated in *C4*.

The slave computes a 64-bit CEDC using the received address and data, and the corresponding bits of this CEDC are compared with the CEDC bits from the master (=1). If these bits match, the write, which was begun earlier when both the address and data became available, is completed and acknowledged. The **ACKNOWLEDGE** is secured by encrypting (X7) and transmitting (T4) a different portion of CEDC generated in step *E2*. This CEDC is encrypted using slave transmission bit stream generated in *C5*. The master verifies the completion of the transaction by decrypting (X8) this portion of the CEDC, using the master reception bit stream from *C6*, and comparing (=2) it with the corresponding, locally generated CEDC bits from step *E1*. As in the secure read transaction, the steps involved in an CEDC comparison can be re-ordered and re-associated, if necessary, to provide faster operation. This re-ordering and re-association may be especially critical at the slave if the CEDC is to be checked and a secure **ACKNOWLEDGE** transmitted on the next bus cycle. This transaction offers a number of opportunities for parallelism, as illustrated in Figure 3-8.

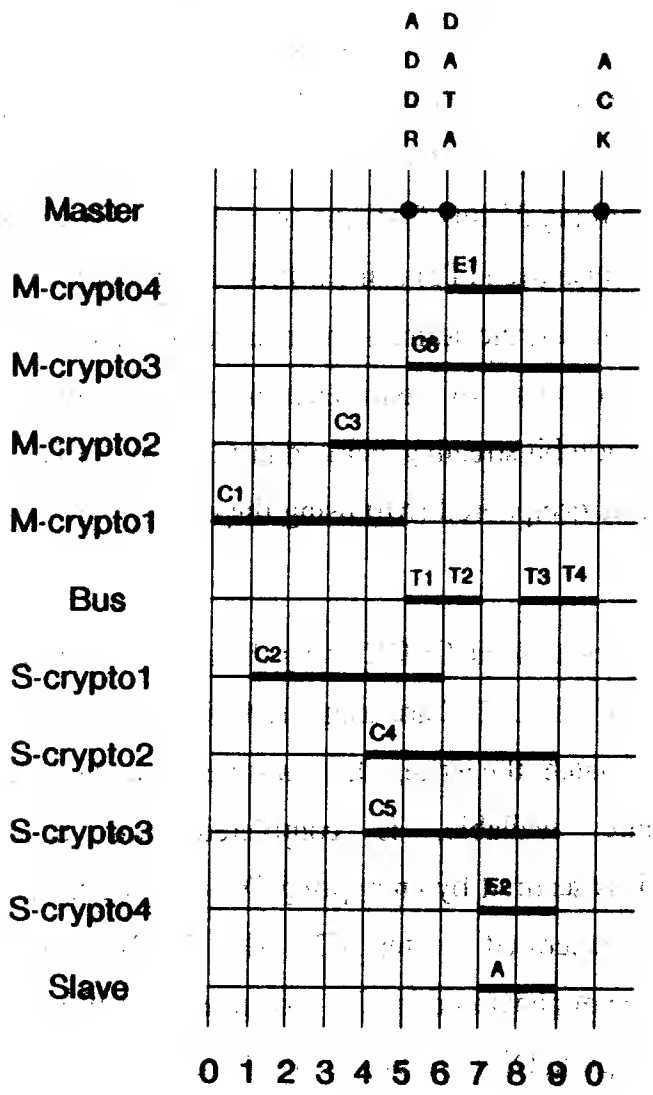


Figure 3-8: Timing Diagram for a **simple secure write**

Total time for this **simple secure write** is $3T_t + T_e$ (5 bus cycles), based in the timing diagram in Figure 3-8, the same as for a **simple secure read**. (An examination of the event graph yields a complex symbolic timing formula, involving nested *minimum* functions, which simplifies to this expression using the relative timing

assumptions adopted earlier.) The address and data are available at the slave at the same points in time as in a **standard write**, but confirmation of their validity is delayed by two bus cycles, causing an equal delay in acknowledgment of the transaction. Again the secure version of this transaction takes 66% longer than the standard version. As an increase in effective memory access time, this delay is not quite so serious as in the case of a **simple secure read** since **write** transactions typically constitute only about 20%-25% of all processor references to memory. Moreover, in systems equipped with a write-through cache, processor-generated **write** transactions may be buffered to reduce the delay associated with access to primary memory. (If a write-back cache is employed, buffering of modified, evicted lines reduces delay on extended write transactions. [6])

Since a **simple secure write** takes 66% longer than a **standard write**, a proportional increase in buffering at the cache will maintain existing performance levels in the face of this additional delay. (A **secure extended write** exhibits the same relative increase in delay.) For cacheless systems, single or double buffering of writes will absorb this delay in most cases. Although additional buffering can reduce the effect of the longer transaction time on effective memory access time for the processor, the transmission of CEDCs during two bus cycles increases bus utilization and thus may delay other transactions. As with **simple secure read** transactions, the problem can be solved by overlapping transmission of CEDCs with address and data transmission (using additional bus lines for this purpose). Use of the extra bus lines and this limited transaction interleaving enables **simple secure write** transactions to proceed at the same maximum rate as **standard write** transactions. Again these bus enhancements are required only for processor-memory transactions and thus affect only **SYSTEM C** and **SYSTEM D**. Using the same reasoning applied to **simple secure read** transactions, it is apparent that **SYSTEM D** is preferred here.

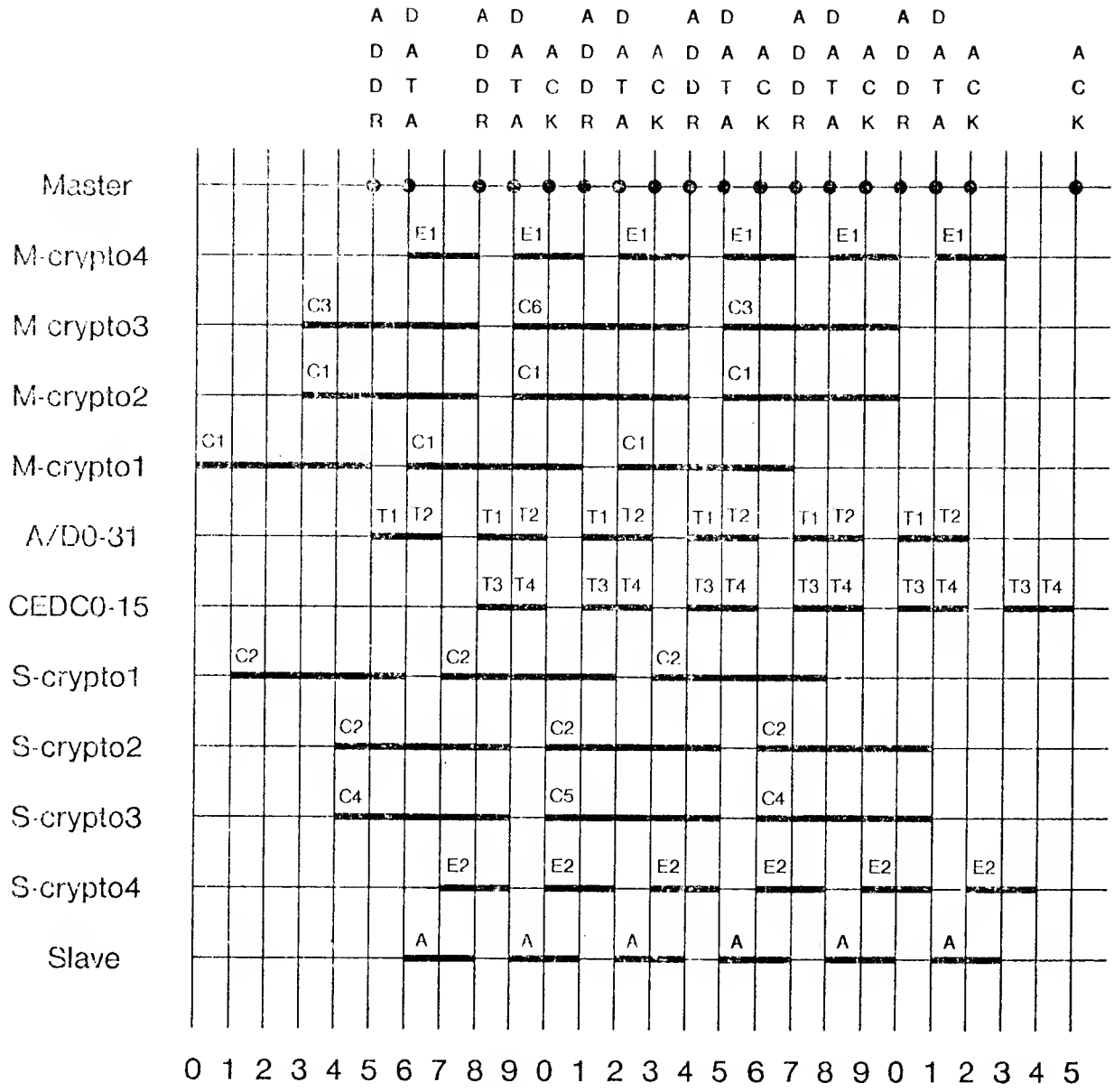


Figure 3-9: Timing Diagram for Successive **simple secure write** Transactions

For processor-memory transactions, the CBIs at primary memory and the memory bus connection to the bus coupler each require four cryptographic devices to maintain the maximum transaction rate, the same number as for a **simple secure read** transaction. Figure 3-9 shows the utilization of memory, bus lines and cryptographic devices for six successive **simple secure write** transactions. The master must conceal 64 bits of address and data and 16 bits of CEDC for each transaction, whereas the slave must conceal only 16 bits of CEDC. Three crypto devices (*crypto1*, *crypto2* and *crypto3*) are devoted to generating bit stream here, with *crypto3* alternating between transmission and reception bit stream generation. (One could make the assignment of crypto devices to bit stream generation tasks simpler by devoting a device exclusively to the slave transmission bit stream, but this would leave two devices idle much of the time.) Again, one cryptographic device (*crypto4*) is required to calculate CEDCs and these CEDCs are transmitted on the extra bus lines **EDC0-15**.

As was the case with **simple secure read** transactions, if addresses need not be concealed then one crypto device can be eliminated. Again, even if addresses are concealed on processor-memory transactions, it seems likely that addresses in control transactions need not be encrypted. Here too, the frequency of **simple secure write** transactions used to control DMA devices should be low enough to allow a single crypto device to generate the transmission and reception bit streams between these transactions, freeing the device to generate the CEDC during the transaction. Thus TRM-packaged peripherals probably require only a single crypto device to keep pace with **simple secure write** control transactions.

3.3.3 Securing *interrupt* Transactions

Only one type of simple transaction has yet to be discussed: an **interrupt**. The security requirements for an interrupt are much like those of a **simple write**, offering

no opportunity for relaxation. The interrupt vector in the the **PRESENT-DATA** must be concealed, and the processor must verify that this operation is properly ordered, authentic and unmodified. The peripheral generating the **interrupt** must verify that the **ACKNOWLEDGE** it receives corresponds to the **PRESENT-DATA** just transmitted. These requirements are readily achieved using the techniques developed above for **simple read** and **simple write** transactions. Figure 3-10 shows the event graph for a **secure interrupt**.

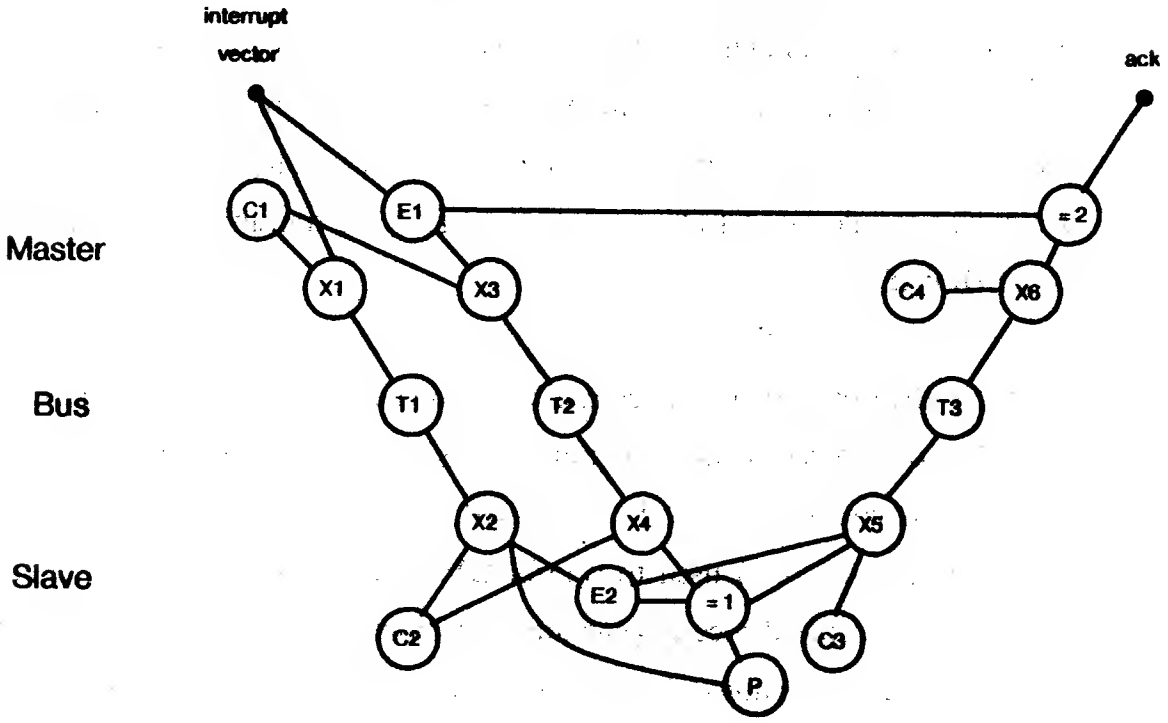


Figure 3-10: Event Graph for a **secure interrupt**

The master begins by generating transmission crypto bit stream to conceal the interrupt vector and CEDC (C1). The interrupt vector is enciphered (X1) and transmitted in a **PRESENT-DATA**. This vector is input to the CEDC calculation

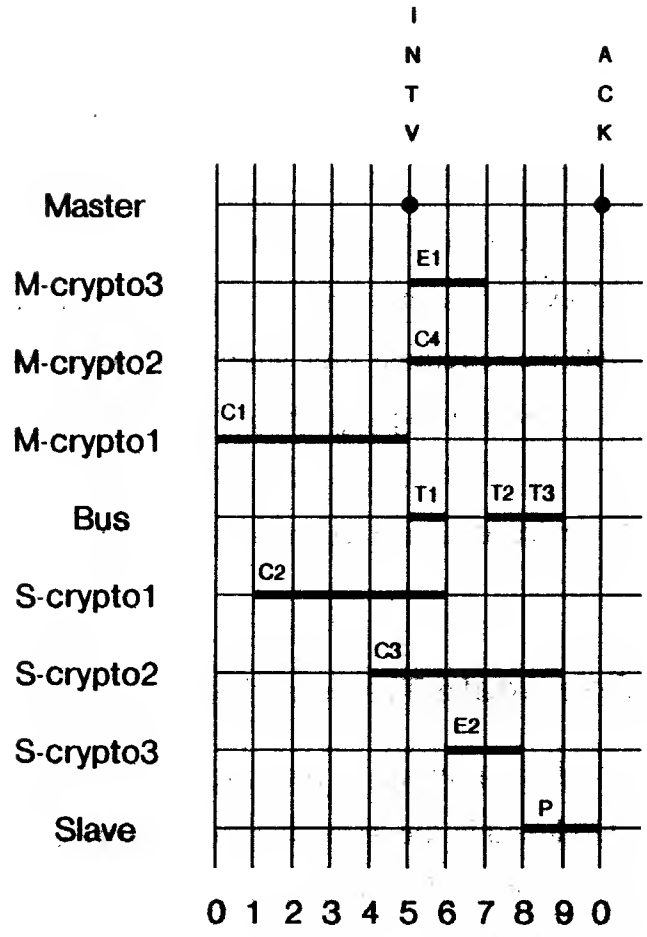


Figure 3-11: Timing Diagram for a **secure interrupt**

(E1) and 16 bits of the result are enciphered (X3) and transmitted (T2). At the slave (processor) the interrupt vector and the CEDC are deciphered (X2 and X4) using the corresponding slave reception bit stream from *C2*. A CEDC is calculated locally on the vector (E2) and the corresponding 16 bits are compared with the transmitted CEDC (=1). If the two values match, the interrupt is processed (P) and acknowledged. The acknowledgment is effected by enciphering another 16 bits of the CEDC (X5) using slave transmission bit stream (C3), and transmitting the result

as an ACKNOWLEDGE (T3). The master deciphers the CEDC (X6) using corresponding master reception bit stream from *C4*, and compares it with the corresponding bits of the CEDC generated locally (=2).

The minimum total time for this transaction is $2T_t + T_e$ (4 bus cycles), based on the timing diagram in Figure 3-11. This is twice as long as a **standard interrupt**, but since these transactions occur so infrequently (they are strictly control transactions), the added delay and extra bus utilization should not significantly affect system performance. The relative infrequency of *interrupt* transactions, like other control transactions, means that a single crypto probably suffices to generate both crypto bit streams and to perform the CEDC calculation. Thus the CBIs for peripheral devices need only one crypto device to handle secure control transactions.

3.4 Securing Aggregate Transactions

This section deals with the problem of securing aggregate transfers. If the simple secure transactions developed in the preceding section were employed for aggregate transfers without interleaving CEDC transmissions (including additional bus lines), utilization of the general purpose or I/O bus for these transfers would increase by 66%. If utilization of this bus is very low, this may be acceptable, but in most cases this increase will noticeably degrade system performance. Adopting interleaving and adding extra bus lines to carry CEDCs, as was done for **simple secure** transactions, is an expensive proposition in this context. This is due to the number of devices attached to this bus and to the fact that this bus is not synchronous, making interleaving more complex. The transactions developed in this section avoid this problem, i.e., they do not significantly increase bus utilization, yet they provide for secure transfers of aggregates between DMA devices and primary memory.

3.4.1 A Transfer Protocol for Data Aggregates

The transfer protocol developed here takes advantage of the fact that transfers between primary memory and these storage devices involve data aggregates larger than a word, e.g., a disk block or a tape record. Rather than checking the validity of each word as it is transferred, the authenticity, integrity and ordering of the aggregate transfer as a whole is checked after the transfer is complete. In this fashion the data and address in each **read** or **write** transaction in an aggregate transfer is encrypted, but the transaction carries no CEDC and thus bus utilization is not affected. Only when the transfer is complete is a cumulative CEDC, covering all of the transferred data and addresses, transmitted for verification. This CEDC transmission is effected using a **simple secure read** as developed in the preceding section.

It might seem that this approach would result in reduced security but a careful examination of the protocol indicates that it presents an intruder with no new opportunities for attacks. When a data aggregate is transferred to primary memory from a storage device, the processor does not access any portion of the aggregate until the storage device signals that the transfer is complete and verified. As long as the unverified data is stored only in the locations that are destined to be overwritten anyway, no real harm results from transferring data aggregates in this fashion. Address filtering of these unverified writes at the slave, restricting them to the region(s) of primary memory which are current targets of such transfers, provides the necessary control. Note that the term *slave* is used here (rather than primary memory) since the filtering and other security functions can be performed at various points depending on system configuration. In **SYSTEM A** and **SYSTEM B** these functions are provided by the CBI in the main TRM and in **SYSTEM D** either the primary memory CBI or the bus coupler CBI (at the I/O bus interface) could perform these tasks.

In transferring data aggregates from primary memory to storage devices a similar argument applies. Some storage devices buffer the aggregate until the transfer is complete, since the rate of arrival of words varies depending on bus traffic and may not be synchronized to the device transfer rate. In this case the aggregate can be checked before it is written on the non-volatile media. Even if the data is written on the media before the transfer is complete (as in a non-buffered device), no harm will result so long as it is possible to identify unverified aggregates on the media. Incomplete transfers to these devices sometimes occur under normal (non-malicious) circumstances due to transmission timing problems. Storage devices (buffered and non-buffered) record an EDC with each aggregate to detect these and other errors. If an incomplete transfer occurs or an error is detected by the cumulative CEDC, the EDC on the media can be set to an error value as a positive indication of unverified data. Since storage devices act as bus masters, there is no need for address filtering here, unlike primary memory.

Thus aggregate transfers to and from primary memory are efficiently and securely implemented using two types of transactions: simple secure transactions to control the transfer, and aggregate secure transactions to transfer the data. The general procedure, for transfers in both directions, is as follows. First, if the transfer is directed to primary memory, the processor identifies the range of the transfer at the slave, i.e., establishes the upper and lower bounds for primary memory references, and resets the slave *cumulative CEDC register*. Next, the processor establishes the transfer parameters at the storage device, e.g., the starting addresses at source and destination and the amount of data to be transferred, using simple secure control transactions. The storage device then carries out the transfer using aggregate secure transactions.

As each word is transferred, the cumulative CEDC is accumulated at both the storage device and at the slave. When the transfer is complete, the storage device

reads the slave control register containing the accumulated CEDC (using a **simple secure read**). In the case of a transfer to memory, this control transaction must set a flag at the slave to prevent further data transfers on this connection until the CEDC register is reset for the next transfer. This value is compared to the CEDC accumulated at the storage device, and the status register at the storage device is set accordingly. (The EDC on the non-volatile media is voided if the comparison fails or if an incomplete transfer error occurs.) The storage device sends a **secure interrupt** to the processor when this procedure is complete and the processor retrieves the contents of the device status register using a **simple secure read.**

Readers who are not interested in the details of securing aggregate transfers should now skip to section 3.7 (page 154) for a summary of the highlights and a review of the conclusions reached in this chapter.

3.4.2 Securing *aggregate read* and *aggregate write* Transactions

The event graphs and timing diagrams for an **aggregate secure read** and an **aggregate secure write** are shown in Figures 3-12, 3-13, 3-14, and 3-15. The encryption/decryption mode and cryptographic error detection techniques employed here are essentially the same as those used in simple secure transactions. The CEDC calculation must be made cumulative in a fashion that not only detects modification of individual words but also detects positional changes (reordering) of words in the data aggregate. The method adopted here is to chain the CEDC calculations by adding the output of the i^{th} CEDC calculation to the input of the $i+1^{st}$ CEDC calculation. This is essentially CBC mode encryption (using a shortened DES) applied to the CEDCs.

In an **aggregate secure read**, the master begins by generating transmission cryptographic bit stream (C1) in the usual fashion. The address in the **PRESENT-**

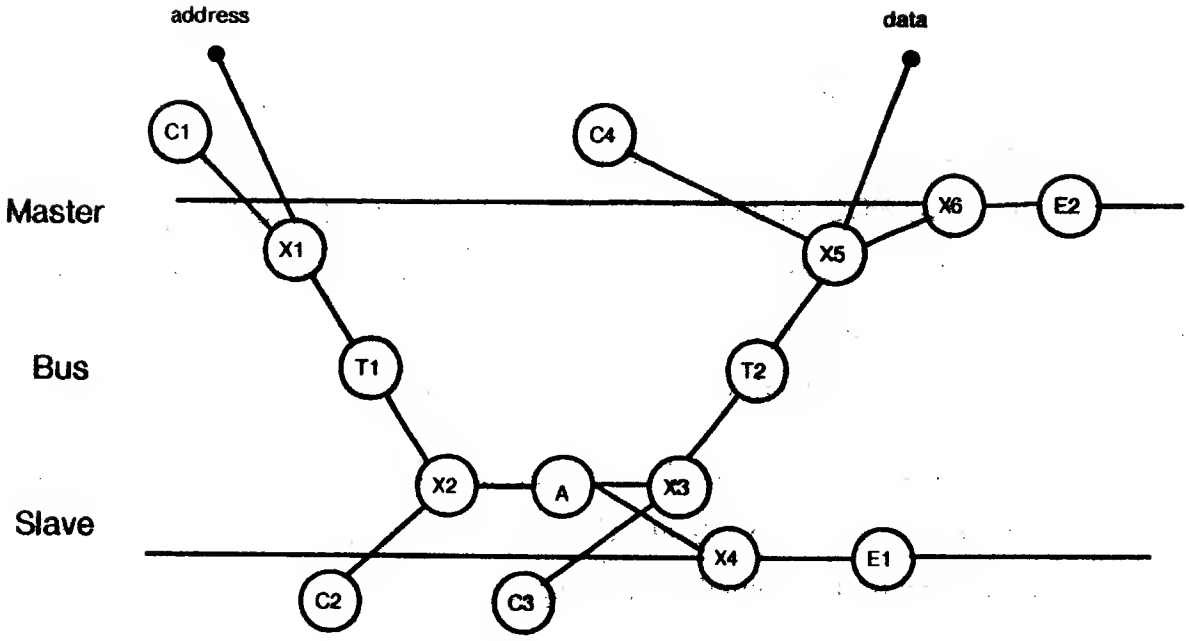


Figure 3-12: Event Graph for an **aggregate secure read**

ADDRESS is enciphered using 32 bits of that bit stream (X1) and transmitted (T1). The slave deciphers the address (X2) using a corresponding portion of the slave reception bit stream generated in *C2*. The appropriate word is retrieved (A), enciphered (X2) using 32 bits of slave transmission bit stream (C3), and transmitted (T2) in a **PRESENT-DATA**. The data is also added to the cumulative CEDC (X4) and a new running CEDC is calculated on the result (E1). At the master, the data is deciphered (X5) using corresponding bits from the master reception bit stream (C4), and is made available both for storage and for calculation of a new cumulative CEDC value (X6 and E2). Figure 3-12 illustrates these processing steps.

An **aggregate secure write** proceeds in much the same fashion. The address in the **PRESENT-ADDRESS** and the data in the **PRESENT-DATA** are enciphered (X1

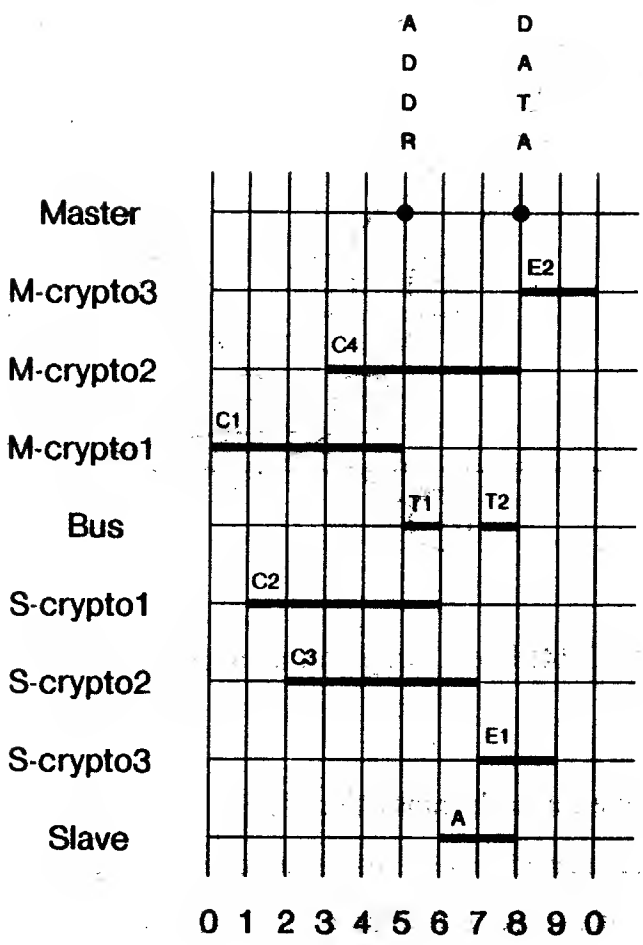


Figure 3-13: Timing Diagram for an **aggregate secure read**

and X2), using 64 bits from the master transmission bit stream (C1), and transmitted (T1 and T2). The data also is fed into the cumulative CEDC calculation (X4 and E1). The slave deciphers the address and data (X3 and X5) using the slave reception bit stream (C2), and transmits an unencrypted ACKNOWLEDGE (T3). The slave checks the address against the range registers (<>) and, if it is within the prescribed bounds, the data is stored and fed into the cumulative CEDC calculation (X6 and E2). Figure 3-14 illustrates these processing steps.

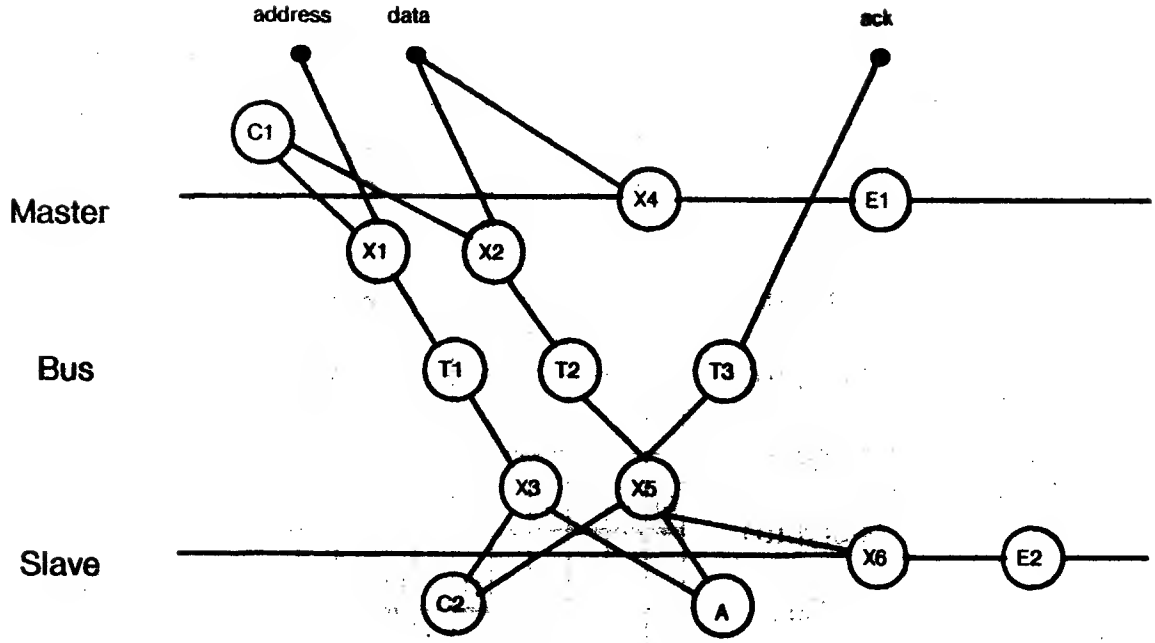


Figure 3-14: Event Graph for an aggregate secure write

The minimum time for both transactions is $2T_t + T_a$ (three bus cycles), the same as for comparable standard transactions, as indicated in Figures 3-13 and 3-15. Note that the CEDC calculation is performed on 64-bit inputs, so it is executed only once for every two transactions. Since the maximum transfer rate for secondary and T&A storage devices ranges from about 1-15 Mbits/s, a single crypto unit probably suffices to generate both the crypto bit stream and to calculate the cumulative CEDC. As it was noted in section 3.3 that a single crypto device is probably sufficient to secure control transactions, this analysis suggests that the CBIs for TRM-packaged secondary and T&A storage require but one crypto device to handle both types of transactions.

This aggregate secure transfer protocol requires an additional two to four control transactions: one to transfer the cumulative CEDC, one to reset the CEDC register at the slave and, in the case of transfers to primary memory, two transactions to

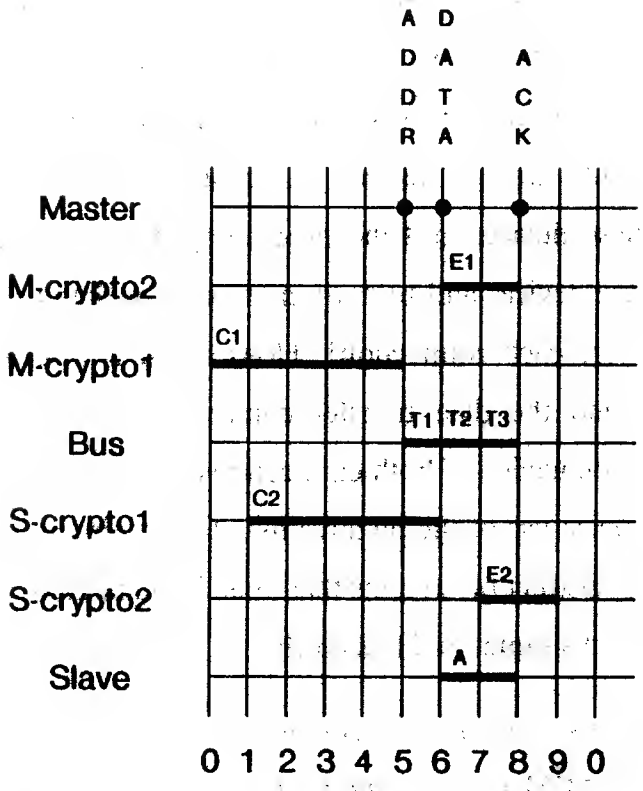


Figure 3-15: Timing Diagram for an **aggregate secure write**

establish the bounds of the transfer. An aggregate transfer in a standard system requires one transaction for every word transferred plus five control transactions (as detailed earlier). Thus, in a typical 512-byte transfer, the additional bus cycles required by extra control transactions to secure the transfer constitute a negligible (1.5-3%) increase in bus utilization for DMA transfers. Moreover, the total time for such transfers is not noticeably increased (<1%) since the extra control transactions require only a few microseconds whereas a 512-byte transfer takes on the order of 500μs at 8 Mbits/s.

3.5 Additional CBI Design Considerations

The cryptographic techniques employed for aggregate secure and simple secure transactions employ a different bit stream ID for each simplex channel, ensuring that the generated bit streams are distinct. In a computer system consisting of *n* TRM-packaged (DMA) storage devices, there are logically *2n* connections: one between each of these devices and the processor (for control purposes) and one between each of these devices and primary memory (for data transfer). This yields *4n* bit streams, two for each connection! However, it is possible to combine the control connection and the data transfer connection for each DMA peripheral device into a single connection if both connections are managed by a single CBI at each end (to synchronize use of the bit streams). Combining these connection pairs halves the number of distinct bit streams that must be generated, making the CBIs at these devices somewhat simpler and less costly.

Combining the control and transfer connections for each device fits naturally in **SYSTEM A** and **SYSTEM B** where the CBI on the main TRM provides the only path to both processor and primary memory for storage devices. In **SYSTEM C** this simplification cannot be effected since the CBIs for primary memory and the processor are distinct in this configuration. However, **SYSTEM C** effectively was eliminated from consideration earlier because of the cost of interleaving CEDC transmission for processor-memory transactions. In **SYSTEM D**, the CBI at the interface to the I/O bus can act as the secure interface to both processor and primary memory for these storage devices in support of combined control/transfer connections. This approach yields single-connection CBIs for secure storage devices, primary memory and the bus coupler interface to the memory bus. Only one multi-connection CBI is needed in these designs, the CBI at the bus coupler interface to the I/O bus.

Irrespective of the choice of combined or separate control and data connections, the above-noted design for **SYSTEM D** is preferred over one in which the primary memory CBI is the termination point for the storage device data transfer connections. The reasoning here is that the primary memory CBI is fairly complex due to the high transaction rate which it must support. If this CBI had to deal with *aggregate* transactions from several storage devices and *simple* transactions from the processor, the bus interface would become even more complex. Thus the preferred design for **SYSTEM D** involves terminating each storage device data transfer connection at the main TRM. Adopting this design, the bus coupler CBI at the I/O bus interface becomes the *slave CBI* in aggregate transfers, and thus it contains the CEDC accumulation register and a pair of bounds registers to restrict access on **aggregate secure write** transactions. Note that these registers are associated with only one transfer at a time so several sets of registers are required to support multiple, simultaneous aggregate transfers.

This is a convenient arrangement since the processor control transactions that manipulate the bounds registers (to establish the range of transfers) do not actually go out on the bus and thus need not be encrypted. Under this arrangement, aggregate transactions are managed at the bus coupler and transformed into simple secure transactions on the memory bus, thus simplifying the primary memory CBI. (In cache-equipped systems configured as **SYSTEM D**, aggregate transfers may store into or fetch from the cache, so these transactions must be decrypted and processed at the bus coupler anyway.) Since the cumulative CEDC detects modification only between the master CBI and the slave CBI, i.e., only on the I/O bus in this design, it is essential that simple secure transactions are used to transport this data on the memory bus.

Using this design, the transfer of a data aggregate between a secure storage device and primary memory involves three distinct phases: transfer on the I/O bus using aggregate secure transactions, buffering in the bus coupler and transfer on the

memory bus using simple secure transactions. On transfers to memory from the I/O bus, a small (two or three word) buffer is usually provided to account for the asynchronous operation of the two busses. If such a buffer were not provided, the time for a store to memory from a device on the I/O bus could double or triple waiting for the memory bus to become available and for an acknowledgment from memory. In the context of an **aggregate secure write** to memory, if this buffer is expanded by one word, the (non-secure) **ACKNOWLEDGE** on the I/O bus can be issued before the **simple secure write** is completed on the memory bus, i.e., the transactions on the two busses can be overlapped.

On transfers from memory to devices on the I/O bus, data is usually pre-fetched from memory into small (one or two word) buffers, one per DMA device. If this pre-fetching were not provided, the time for a fetch from memory by a device on the I/O bus could double or triple, just as for stores by these devices. In the case of an **aggregate secure read**, the size of these buffers need not be increased, even though a **simple secure read** encounters a two-cycle delay before the authenticity, integrity and timeliness of the transmitted data is verified. Instead, the prefetch can begin two cycles earlier than in a standard system so that the requested word is available and checked before the **aggregate** transaction takes place. If the same prefetch time were employed, the data from primary memory might not be checked before it was transmitted on the I/O bus and thus the entire transfer would have to be aborted if the check on the word failed. Earlier prefetching is readily accomplished by the bus coupler given the relatively low transfer rates of storage devices on the I/O bus. To avoid pre-fetching past the end of the data to be transferred, one can use the bounds registers provided for **aggregate secure write** transactions to delimit the range of the transfer on **aggregate secure read transactions.**

One final design requirement that arises in all system configurations is the need for CBIs on the general purpose or I/O bus to be able to determine when transactions are directed toward them. This is a problem here because all addresses in secure transactions are encrypted and can only be decrypted using the proper crypto bit stream. (Of course, if the system designer elects not to encrypt addresses this problem vanishes.) It is conceivable that a CBI attempting to decrypt an address using the wrong crypto bit stream will yield a value that matches an address at the CBI. The multi-connection CBI at the bus coupler would be further complicated if it had to check the address in each transaction to determine the connection with which it was associated. There are dual problems here: secure storage device CBIs need to know whether they are the *target* of a transaction whereas, the main TRM CBI (on the I/O or general purpose bus) needs to know the *source* of a transaction. Note that the problem is symmetric but not identical for the main TRM and for storage devices. Based on the data flow patterns encountered in these systems, if the main TRM is not the source of a transaction it must be the target, and if a device is the target, then the main TRM must be the source.

If the arbitration procedure on the I/O or general purpose bus explicitly identifies the next transmitter (the next source), then the second problem is solved, i.e., the source of each transaction is identified for the main TRM CBI. Moreover, using this information, the storage device CBIs know they are not the target of a transaction if the source is not the main TRM. The only remaining problem is identifying the target of control transactions issued by the TRM. If the addresses in these control transactions are not encrypted, the target is clearly identified and no confusion results. In most applications, this will not be regarded as a serious breach of security, as noted earlier, since only the addresses of control registers are involved and these provide little traffic analysis information. If the arbitration procedure does not identify the next transmitter, the CBIs on the I/O bus can generate this information and transmit it using some additional bus lines. About two or three additional bus lines should suffice for this purpose.

3.6 System Integration Issues

The preceding sections dealt with the problems of securing communication involving the processor, primary memory and secondary and T&A storage devices. Although these problems are central to the design of computer systems that achieve the security requirements outlined in section 3.2, some additional problems must be be addressed to complete the design. For example, there also has been no discussion of how to interface non-secure devices to the I/O bus so that they can communicate with the processor and, in the case of DMA devices, with primary memory. System initialization procedures, responses to possible security violations and enforcing reloading constraints associated with archival storage are all topics requiring further attention. The remainder of this chapter deals with each of these topics in turn.

3.6.1 Interfacing Non-Secure Devices on the I/O Bus

The non-secure devices attached to the general purpose or I/O bus fall into two classes: interrupt driven and DMA. Interrupt driven devices interface only with the processor, generating interrupt transactions and acting as the target of read and write transactions to device control registers. DMA devices exhibit the same processor interface requirements and further require a means of transferring data aggregates to and from primary memory. Secure and non-secure devices must co-exist on the general purpose or I/O bus without either being *confused* by the addresses transmitted by the other. In solving these interface problems it is most desirable to avoid approaches that entail modifying the bus interfaces for non-secure devices. This is an important consideration since there may be a number of these devices on the I/O bus, and system cost might increase significantly if off-the-shelf versions of these devices cannot be employed.

First consider the problem of transmitting both encrypted and clear addresses on the general purpose or I/O bus. Since the bit pattern that results from encrypting an address is unpredictable, it is conceivable that some encrypted addresses will match the bus addresses of non-secure devices and, conversely, that clear addresses could be *decrypted* by secure devices to yield spurious bus addresses. In section 3.5, two solutions were presented for resolving an analogous problem resulting from the ambiguities presented by encrypted addresses used on different connections. One solution, the use of extra I/O bus lines to identify the transmitter and destination of bus transactions would solve the current problem as well, but this would violate the goal of not modifying the bus interfaces of non-secure devices. The other solution, based on using clear addresses in control transactions and an arbitration scheme that identifies the transmitter, also requires that bus interfaces (other than the processor) know not to perform address recognition except when the processor is the transmitter.

To avoid any modification of non-secure bus interfaces, the strategy proposed for bus address assignments in the monolithic TRM design is adopted here. The high order bit of addresses will be used to distinguish between secure and non-secure device addresses and this bit will not be encrypted in *any* operations on the general purpose or I/O bus. This bit partitions the bus address space between secure and non-secure devices, so neither type of device will be confused and no modifications to non-secure device bus interfaces are required. Since this address bit merely identifies which type of device is being addressed, any traffic analysis information gleaned from examination of this bit would be readily available in any case. Note that this bus address assignment strategy does not interfere with use of either of the previously mentioned solutions to the encrypted address ambiguity problem as it exists among secure devices.

Using this address assignment scheme, interfacing non-secure interrupt driven devices becomes fairly simple. These devices generate **standard interrupt** transactions and the processor controls the devices using **standard read** and **standard write** transactions. The fact that the high order bus address bit distinguishes between non-secure and secure devices means that the processor implicitly indicates to its CBI whether or not a transaction should be encrypted. In the case of stores by non-secure DMA devices, there is a need for address filtering to restrict access to designated memory locations. This is accomplished using pairs of bounds registers, as proposed earlier for the secure bus coupler (SBC) in the monolithic TRM design. The processor must establish the range of memory locations to be accessed by non-secure DMA devices and indicate the allowed modes of access (fetch and/or store) before transfers can proceed. If an arbitration mechanism is employed that identifies the transmitter, the appropriate pair of bounds registers is trivially selected, otherwise an associative search (based on the address in the transaction) may be required.

3.6.2 System Initialization

In the preceding sections, secure operation of the computer system has been described in a *steady-state* context. When the computer system is powered up or otherwise periodically initialized, it is necessary to establish the context for secure, steady-state operation. The purpose of this initialization procedure is the establishment of secure connections between the main TRM and the other (*slave*) TRMs in the system. The requirements for secure connection initiation here are the same as in general purpose communication environments, i.e., the authenticity and the time-integrity of each connection must be established. The methods for achieving these requirements are somewhat simpler here due to the fixed connectivity patterns of the TRMs and due to the fact that there is no mutual

suspicion among the TRMs. The initialization procedure involves distribution of a *working key* by the main TRM followed by a challenge-response protocol to verify the authenticity and time-integrity of the connection.

Each slave TRM contains three non-volatile control registers for security purposes: one contains the master key of the TRM, one holds a bit pattern used in the challenge-response protocol and one records the bit stream ID pair used by the TRM in communicating with the main TRM. One volatile register, to hold a working key, is also included in each slave TRM. The registers containing the master key and the challenge-response value are both loaded at the time of manufacture, and the master-key register is never changed. However, the registers containing the challenge value and the bit stream IDs are modified each time the TRM is reset (using the bus **RESET** line). The main TRM contains a collection of non-volatile registers, including one for its master key, a counter for generating working keys and a set of registers to hold the master keys and bit stream IDs for the slave TRMs configured in the system. The master keys of slave TRMs are loaded into the main TRM using a procedure described in section 3.6.4. The main TRM generates new working keys by incrementing its non-volatile counter and encrypting (using ECB mode) the result under its master key, generating a distinct, unpredictable working key each time. System initialization proceeds as follows.

First, the main TRM generates a new working key as described above. Next, for each slave TRM in turn, the main TRM raises the **RESET** line while asserting the bus address of the TRM being initialized, clearing all volatile registers in that slave TRM. The main TRM then enciphers the working key under the slave TRM master key (using ECB mode) and transmits the result to slave TRM control registers using two **standard write** transactions. The slave TRM receives the working key, deciphers it (using the slave TRM master key) and loads the result into its (volatile) working-key register. Next, the master TRM uses a **standard write** to store the assigned bit

stream ID pair to the slave TRM. The master TRM chooses these IDs so that each slave TRM uses a different pair to communicate with the main TRM. The master TRM also stores these values (working key and bit stream IDs) into the CBI registers it associated with the slave TRM being initialized.

Using its master key, the slave encrypts the contents of its challenge-value register, yielding a new challenge value. The counter(s) used to generate crypto bit stream are initialized appropriately, i.e., the counter for a single crypto device CBI is set to *1*, and if *n* crypto devices are used, their counters are set to the values *1* through *n*. The slave TRM then generates a **secure interrupt**, using the new working key and the assigned bit stream IDs, indicating that it is prepared to carry out the challenge-response protocol. The main TRM responds by reading the *challenge-value* register and then writing back the value, using **simple secure** transactions. The ability of the slave to generate a valid **secure interrupt** using the new working key verifies the authenticity and time-integrity of the connection to the main TRM, whereas the successful reading and writing of the challenge-value register does the same for the slave TRM. When this procedure has been carried out for all slave TRMs, the system is initialized for secure inter-TRM communication.

3.6.3 Response to Potential Security Violations

The CBIs and the TRM operating system detect potential security violations in two ways: through mismatches between calculated and received CEDCs and through timeouts. Each time a violation is detected at the main TRM, a non-volatile *violation counter* is incremented to record the occurrence. This type of *threat monitoring* is used to detect attempts by an attacker to subvert the protection mechanisms by repeated trials. A threshold is established by the vendor and, if that threshold is exceeded, the processor will shut down (refuse to execute external software for the client) until the vendor intervenes. This intervention may involve

an inspection of the system by a representative of the vendor, or it may simply require network communication so that the vendor is appraised of the repeated errors. The main TRM may be reset by engaging some form of dialogue with the vendor, analogous to the system initialization procedure described above.

Violations are detected at the bus master and at the slave, depending on the type of transaction and the type of violation. The violations may result from transmission errors on the bus (accidental or malicious), loss of cryptographic bit stream synchrony between communicating CBIs or because of a transient or "hard" device malfunction. A simple parity check is used to detect non-malicious errors in data, addresses or interrupt vectors on bus operations (bus lines **PARITY0-3**), and it is expected that this code will catch most such errors. If a bus operation fails this non-secure error detection code test, the operation is retransmitted automatically and the violation counter is *not* incremented. (This *operation retransmission* uses a buffered value of the operation and should not be confused with the *transaction retry* described below.) Only those "errors" detected by the CEDC or by a timeout are treated as attempted security violations. The *appropriate* response to a violation depends on the type of violation, the type of transaction and whether the slave of master detects the violation.

First consider CEDC mismatches. In the case of a **simple secure read**, this type of violation is detected at the master CBI and the response is to attempt the transaction again, treating it as a new transaction from the standpoint of the security measures. Thus new cryptographic bit stream is generated for the retried transaction. In the case of a **simple secure write** or a **secure interrupt**, the violation is detected at the slave and the response is to ignore the transaction, allowing the master to timeout waiting for the **ACKNOWLEDGE**. For *aggregate secure* transfers (stores and fetches), the DMA storage device determines if the cumulative CEDC check fails, and the operating system discovers the violation when it fetches the control register

from this device. The operating system, upon detecting this condition, increments the violation counter and may retry the aggregate transfer.

Next consider the response to timeouts. In the case of a **simple secure read**, a timeout occurs at the master CBI when either the data or the CEDC fails to arrive. The response is to discard any cryptographic bit stream generated for this transaction and retry the transaction, treating it as a new transaction. In the case of a **simple secure write** or a **secure interrupt**, a timeout can occur at either master or slave CBI, e.g., while waiting for the CEDC or the **ACKNOWLEDGE**. If the slave experiences the timeout, it ignores the transaction and discards any cryptographic bit stream for the transaction. If no **ACKNOWLEDGE** is received, the master will timeout, so all timeouts on these transactions are translated into timeouts at the master. The master discards the cryptographic bit stream associated with this transaction and retries it. In the case of aggregate transactions (fetches or stores), timeouts are handled as above, noting that the cumulative CEDC is not updated on the retry.

If the retry fails in any of these cases, it is necessary for the operating system to handle the situation. In the case of *simple secure* transactions, the processor is the master and will detect the problem when the retry fails. The processor readily detects failed **secure interrupt** transactions as well. In the case of *aggregate secure* transactions, the secure storage device will send a **secure interrupt** to the processor to signal the error. Either way the operating system is easily notified of the problem. The only recourse for the processor is to reset and reinitialize the device (establishing a new bit stream ID for the CBI) to rectify possible cryptographic bit stream synchrony problems or to detect an inoperative device (identified by its lack of response to the initialization procedure). If this procedure succeeds it may be possible to recover from the point at which the failure occurred. (An aggregate transfer would have to be retried in its entirety.) If the procedure fails it is time to call the vendor.

3.6.4 Distributing TRMs and External Software

TRM distribution arises in two contexts: distribution of external software by TRM-packaged transfer storage and additions of TRM-packaged devices to systems. The same hardware distribution procedure is employed in both contexts. The vendor maintains a database that contains the serial number, master key, and initial challenge-response value for each TRM he has manufactured. Given the serial number of a slave TRM to be added to a system and the serial number of the main TRM for that system, the vendor can use this database to generate a bit string that is entered into the main TRM of the system in question (via a terminal). This bit string consists of the initial challenge-response value and the master key for the slave TRM being sold, both encrypted under the master key of the main TRM (using PCBC mode). When a client purchases a TRM-packaged device to add to his system, the local vendor representative contacts the vendor computer that maintains the database described above, transmits the requisite serial numbers and receives this bit string in response. In this fashion a main TRM acquires master keys for slave TRMs. This method does not impose long delays as the factory customizes TRMs for specific systems nor does it require trust in the local vendor representative!

Physical transfer storage may not be implemented in the encrypted bus approach because of the high cost of TRM packaging for demountable storage media. Instead, external software will most likely be distributed via a communication network as described in section 2.3.4. However, one can develop mechanisms for distributing external software via transfer storage media. These mechanisms are not directly related to the encrypted bus techniques developed in this chapter but rather are based largely on operating system conventions. For transfer storage, there is a requirement that related files (transfer units) on this media be loaded into the file system on secondary storage together and that the operating system be able to

distinguish between vendor-supplied (external) software and client-written software. Moreover, since the client may use transfer media as archival storage for external software, any reloading constraints associated with files in transfer units must be checked when loading these units into the file system.

The following operating system mechanisms achieve these requirements. All TRM-packaged, demountable storage media must contain a header (not accessible by client I/O operations) that identifies the type of storage on the media (secondary, transfer or archival). The operating system checks this header when the media is mounted, preventing any confusion as to what type of files are contained on the media. Each transfer unit is recorded as a file consisting of a table of contents and a list of any non-reloadable files contained in the unit followed by the files that make up the transfer unit. The operating system loads all of the component files of a transfer unit into the file system together, deleting any existing copies of these files. (Existing copies of these files are deleted to ensure the consistency of the transfer unit in the file system, i.e., to prevent mixing of files from old and new releases of external software.) The only exception is that any non-reloadable files in the unit are not loaded if they exist or if they have existed previously (as explained in the next section). These mechanisms are quite similar to those employed in the encrypted storage approach for securing transfer storage (see section 4.3).

3.6.5 Secure Archival Storage Reloading Constraints

In section 2.1 three classes of files were distinguished with respect to the constraints placed on reloading these files from secure archival storage into the file system on secure secondary storage. A client may be free to reload any copy of a file (*unconstrained*), he may be allowed to reload only the most recent archived copy of the file (*most-recent-only*) or the file may be declared *non-reloadable*. There also may be a requirement that reloadable files be grouped into archival units, so that all

of these files are reloaded together. Archival storage is presumed to be demountable and, as with transfer storage, it is not clear if demountable media can be TRM-packaged in an economically feasible fashion. Thus the problem of enforcing reloading constraints may never arise in systems based on the encrypted bus approach. However, one can outline a method of enforcing these constraints in the context of such systems. The method proposed here, like the one described above for transfer storage, is based on operating system conventions for saving and reloading files from archival storage. These conventions depend on the maintenance of a table that identifies non-reloadable files and that lists the name and the time and date of the most recent copy of files archived with that reloading constraint.

All files on archival storage are represented as archival units using the same type of format as transfer units, i.e., a table of contents of the files contained in the unit, the reloading constraint associated with these files and the time and date the unit was written. (All of the files in an archival unit share the same reloading constraint.) The operating system provides a mechanism by which external software can direct (automatically or in response to a client request) one or more files to be saved as an archival unit along with the reloading constraint for the unit. The operating system also maintains a directory on each archival storage volume for locating files in archival units on that volume. A request to reload a file causes all of the files in the unit to be reloaded, subject to the reloading constraint associated with the unit. Non-reloadable files are so marked on secondary storage by the operating system and thus are not subject to archiving.

The operating system maintains a table on (non-demountable) secondary storage identifying all non-reloadable files and listing the time and date when the last archival unit containing each file with the most-recent-only reloading attribute was written. This table is consulted when a unit with the most-recent-only constraint is

reloaded, when transfer units containing non-reloadable files are loaded or when external software requests creation of a non-reloadable file. If this table is destroyed, no files with the most-recent-only reloading constraint can be reloaded and no non-reloadable file can be created or loaded from transfer units. Thus this table must be maintained in a highly reliable fashion. Section 4.3.4 describes techniques for ensuring the robustness of an equivalent table used for the same purpose in the encrypted storage approach and these techniques are applicable here. The interested reader is referred to that section for further details.

3.7 Conclusions

The techniques developed in this chapter enable a computer system constructed from two or more TRM-packaged pieces to protect external software from disclosure and undetected modifieation. Several important techniques were introdueed in this chapter. The stream cipher mode employed here is specially designed to minimize delay and maximize throughput. In particular, this mode permits multiple crypto devices to be used in parallel to generate crypto bit stream at very high rates. The shortened DES calculation employed for CEDCs enables *simple secure* transactions to proceed at relatively high rates. Use of a distinct crypto bit stream for each simplex channel supports asynchrony in secure transaction scenarios. This is critical to the elimination of authentication checks at the slave during *simple secure read* transaetions (enhancing throughput) and it allows control and data transfer connections to be combined. Finally, *aggregate secure* transactions reduce overhead on data transfers between primary memory and TRM-packaged storage devices by transmitting a cumulative CEDC at the completion of the transfer, rather than transmitting a CEDC with each transaction.

The only weakness of the designs presented in this chapter arises from the limited traffic analysis that can be carried out on exposed portions of the bus. The amount of information that is released in this fashion depends on the choice of configuration, but it is very small in most cases anyway. In **SYSTEM A** and **SYSTEM B** the impact of the protection measures on system performance is negligible and the cost of the required CBIs should be acceptably small. For systems in which primary memory is independently packaged, the performance impact of these measures is greater, but this impact can be minimized through appropriate configuration choices, e.g., a cache-equipped, dual-bus design. Thus **SYSTEM D** is preferred over **SYSTEM C** since the dual-bus design minimizes the cost of proposed bus enhancements and yields simplier CBIs. However, the processor and memory CBIs in both systems may be expensive, due largely to the number of cryptographic devices required.

Demountable media could be developed for these designs, but it is not clear if such media would be economically feasible to produce, since both the media and its access hardware must be packaged together. Thus distribution of external software is best accomplished through secure communication techniques as described in section 2.3.4 and demountable secondary or archival storage options may be limited or non-existent. The encrypted bus designs offer greater flexibility than the monolithic TRM design, but the cost of TRM packaging, including CBIs, may preclude the configurations that offer the greatest flexibility, e.g., **SYSTEM D**. The encrypted bus approach is highly transparent, i.e., there is little or no impact on most external software and very little software is devoted to managing the protection mechanisms. By adopting appropriate conventions for assignment of bus addresses, CBIs can determine if a transaction should be repeated outside the TRM and, if it is repeated, whether it must be encrypted.

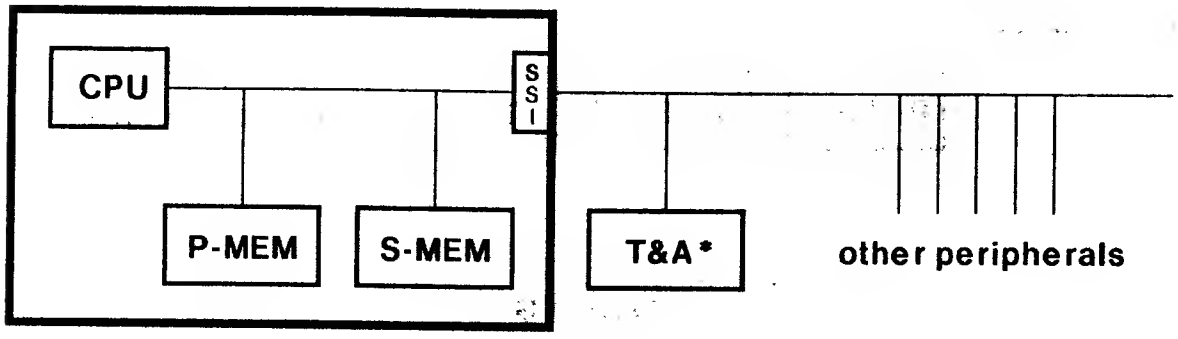
Chapter Four

An Encrypted Storage Approach to Protecting External Software

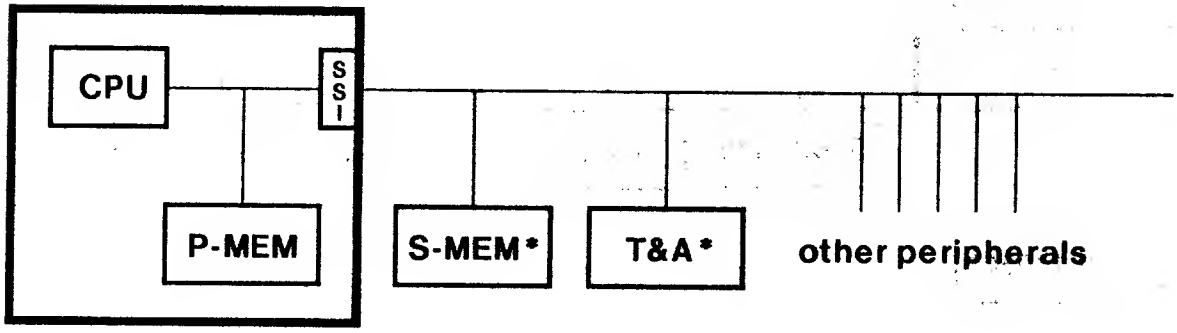
This chapter explores in detail an approach to securing external software based on the use of cryptographic and protocol techniques to protect data stored outside a TRM (using physically unprotected media and devices). In this approach, a processor and some of the *lower* levels of the storage hierarchy are enclosed in a single TRM and all data in *higher* levels of storage (outside of the TRM) are protected by being encrypted and by the use of appropriate protocols. This design approach allows significant use of off-the-shelf equipment since the storage and transmission of encrypted data is generally transparent to the devices and the bus(es). Special equipment is required only at the point where data must be cryptographically transformed, i.e., at the TRM boundary. These transformations are effected by a *secure storage interface* (SSI) that provides encryption, decryption and error checking services.

The boundary between the TRM and physically unprotected storage occurs at one of three points, as illustrated in Figures 4-1 and 4-2. In SYSTEM E only transfer and archival storage is outside the TRM, whereas in SYSTEM F secondary memory is also physically unprotected and in SYSTEM G and SYSTEM H even data in primary memory is subject to intruder attack. These four system configurations correspond directly to those presented at the beginning of Chapter 3. Here too the organization of the processor and primary memory (dual or single bus system, cache or cacheless processor) are irrelevant in the first two systems (E and

F). In the latter two systems (G and H) the choice of a single or dual bus arrangement and a cache or cacheless processor is critical.

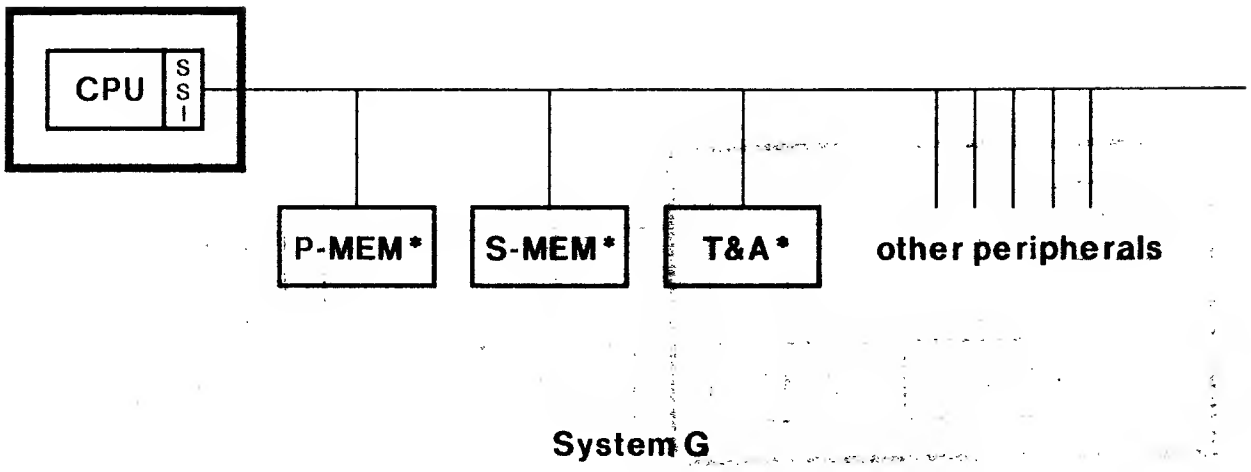


System E

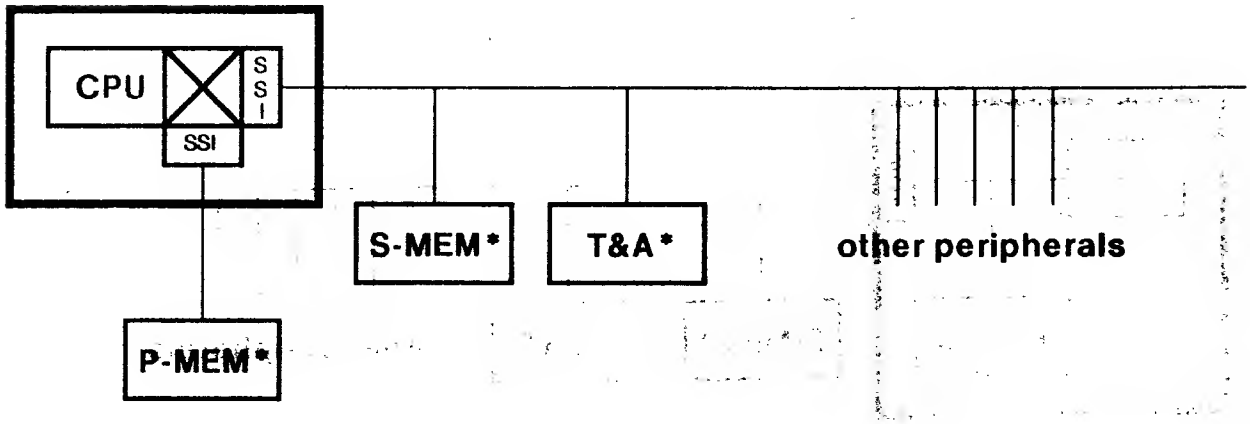


System F

Figure 4-1: Two System Configurations Employing a TRM and an SSI



System G



System H

Figure 4-2: Two More System Configurations Employing a TRM and an SSI

As in Chapter 3, successive configurations decrease the number of devices contained within a TRM, increasing flexibility by allowing more options in equipment selection and greater opportunity for system change both for growth and maintenance. Here, since only one TRM is employed, these configurations allow for even greater flexibility since devices outside the TRM are off-the-shelf. These designs make practical the use of conventional media for T&A storage and demountable secondary storage, overcoming a serious limitation of the encrypted bus designs. Moreover, these designs use fewer TRMs and encryption chips, thus reducing overall system cost as compared with the encrypted bus approach. These improvements are not without attendant costs. The encrypted storage approach requires explicit software control by external software or operating systems to manage databases that are part of the protection mechanisms. These databases decrease available storage at each level in the hierarchy and require maintenance activities that involve additional transfers among levels in the storage hierarchies (resulting in processing delays and decreased bus availability).

4.1 Security Requirements in the Encrypted Storage Approach

The two major aspects of protecting external software, preventing release of and detecting modification of information, translate into several specific requirements in the context of encrypted storage designs. In this context storage devices and bus segments outside the TRM are subject to physical attack by an intruder and the semantics of secure operation are somewhat different from those encountered in the encrypted bus environment. Thus, instead of defining secure system operation in terms of individual bus transactions, here system security is defined in terms of reading and writing of *storage units*, encrypted collections of data that are independently protected. This higher level specification of security requirements encompasses attacks launched against vulnerable bus segments and storage devices.

Figure 4-3 shows the simple model used to discuss intruder attacks and security requirements for encrypted storage designs. This model applies to all four configurations shown in Figures 4-1 and 4-2. Only two operations, **Read** and **Write**, are included in this model. These operations transfer storage units across the boundary between protected storage in the TRM and unprotected storage outside the TRM. Note that several bus transactions are usually required to effect these higher level operations, e.g., transfer of a disk sector between primary and secondary memory involves control transactions and a number of **read** or **write** transactions to effect a storage unit **Read** or **Write**. Each operation involves two values: the *storage unit* being transferred and an *identifier* (ID) that designates the storage unit. (The size of the storage unit is either implicit or derivable from the representation of the unit.) Different storage units and corresponding IDs are employed for each level in the memory hierarchy.

In transfer and archival storage the units are collections of (one or more) logically inter-related files that are distributed or archived and reloaded together (see section 2.1). In this context IDs are often character string file names, perhaps qualified by the date and time at which the storage unit was created. In secondary memory the storage units are generally disk sectors and the IDs are sector addresses qualified by disk identifiers. Files do not fit the definition for storage units at this level in the memory hierarchy since individual sectors may be read or written and processed independently of other portions of the file and since non-file data structures, e.g., directories and file maps, also must be be protected. In primary memory there are two choices for storage units, words and cache lines, depending on processor configuration. Because of the space overhead associated with each storage unit for security purposes (described in the following sections), cache lines offer the only practical option for storage units in primary memory. In this context, IDs are primary memory addresses truncated to reflect the size of cache lines.

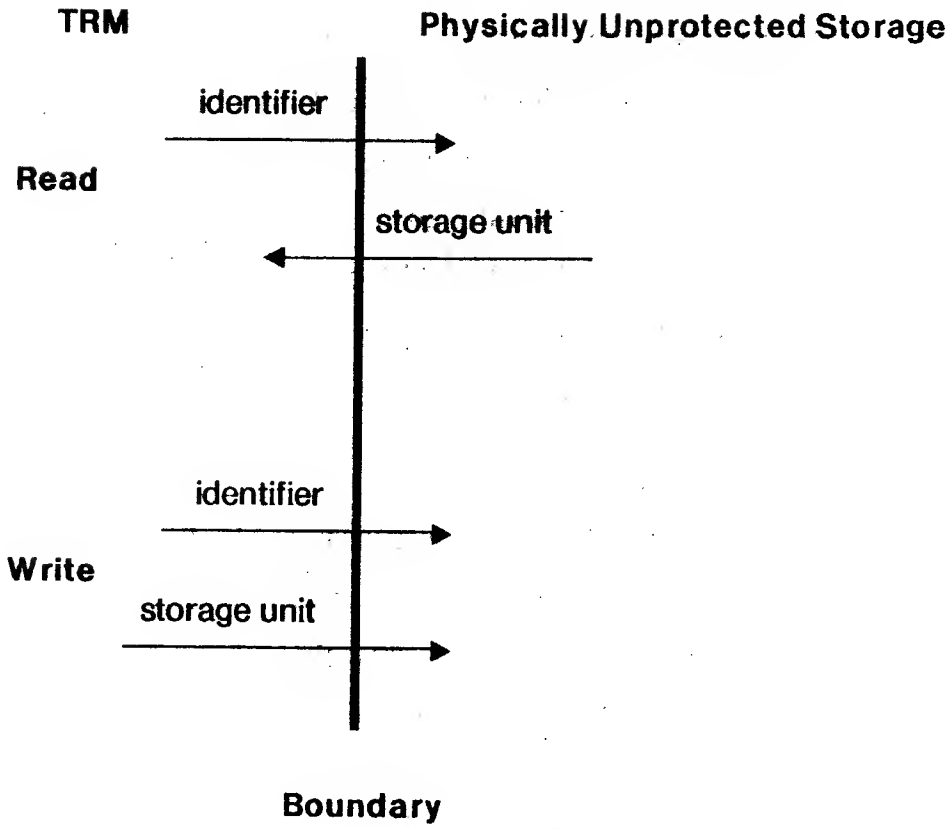


Figure 4-3: A Simple Model for Encrypted Storage Operations

Using the model pictured in Figure 4-3, the vulnerabilities and corresponding security requirements for **Read** and **Write** operations are readily stated. In a **Write** operation both the storage unit and its ID are transmitted by the TRM across the boundary. Unless suitable precautions are taken, the data in the storage unit will be exposed to an intruder. Hence concealment of data in the storage unit, including hiding of patterns within and across storage units, is an obvious requirement.[9] An

[9]Note that a **Write** to a secondary or T&A storage device is effected through **read** bus operations (directed to primary memory) by that storage device. Thus there is an additional requirement that these **read** operations be restricted to appropriate primary memory locations.

intruder also can effect information release by engaging in traffic analysis, i.e., by examining patterns of access to physically unprotected storage. The ID associated with each operation cannot be concealed; it must be available so that devices can correctly store and fetch the storage units. Therefore some level of traffic analysis is always possible using this approach. As in the encrypted bus approach, the amount of information available through traffic analysis is configuration- and application-dependent. In general, **SYSTEM E** provides fewer opportunities for traffic analysis than **SYSTEM F** which in turn provides fewer than **SYSTEM G** or **SYSTEM H**. Each of these configurations provides more detailed traffic analysis information than the corresponding encrypted-bus configuration.

In a **Read** operation, an ID is transmitted by the TRM across the boundary and the physically unprotected storage system returns a storage unit. Thus **Read** operations release information only through traffic analysis.[10] The remaining security requirements for **Read** operations deal with detecting modification of information and are simply explicit statements of the assumptions usually associated with *normal* system operation. Thus the requirements associated with a **Read** are simply stated: The storage unit returned in response to the **Read** must be the most recent unit written by the TRM using the same ID specified in this **Read**, and the unit must not have been modified while outside the TRM. This concise statement embodies the authenticity, integrity and timeliness assumptions implicit in normal operation.

The timeliness assumption is important since it is the foundation upon which various application-specific consistency algorithms are constructed, especially at the primary and secondary storage levels. If software executing in the TRM could not

---

[10]Note that a **Read** from a secondary or T&A storage device is actually effected through bus **write** operations (directed to primary memory) by that storage device. Thus there is also a requirement to restrict those **write** operations to appropriate primary memory locations.

be certain that the disk record or cache line just read was the last one written with the same ID, secure operation would be impossible! However the timeliness guarantee is not so well suited to transfer and archival storage. For transfer storage, the guarantee is not applicable since this storage is, by definition, externally supplied and not modified by the TRM. (The assumption here is that these storage units consist of programs and associated static, immutable databases.) Here consistency is expressed by grouping files into transfer units (see sections 2.1 and 3.6.4). For archival storage, consistency is expressed by grouping files into archival units and by the reloading constraints associated with files. For archival storage, a timeliness guarantee is required in some cases (most-recent-only and non-reloadable files) and may be ignored in others (unconstrained reloading).

This perspective of intruder attacks and corresponding security requirements views **Write** operations as subject to attacks that release information (directly or via traffic analysis) whereas **Read** operations are subject to traffic analysis and to various modification attacks. More precisely, modification attacks during **Write** or **Read** operations or while data is held in storage are detected only at the time when the modified storage units are transferred (by a **Read**) across the boundary into the TRM. The model does not distinguish when or where a modification attack occurs, e.g., on the bus during a **Write** or **Read** or in the interim when the data is in storage. This level of abstraction in discussing attacks and defining requirements is appropriate since the protection mechanisms developed in this section counter these attacks independent of the fashion in which they are effected. In addition to these requirements for operations on encrypted data, there is the need to restrict access to locations within the TRM (primary memory and device control registers) by non-secure DMA devices, a requirement that also arose in the encrypted bus approach. The next section refines this description of security requirements and presents techniques selected for meeting these requirements.

4.2 Basic Techniques for the Encrypted Storage Approach

A combination of cryptographic and protocol techniques are employed to achieve the requirements established in the preceding section. Although these techniques vary slightly depending on system configuration, the basic concepts involved are the same in each case. One type of attack, traffic analysis, is essentially identical in both encrypted bus and encrypted storage environments and is treated in essentially the same fashion in both. In both environments the only way to counter such attacks is through the generation of sufficient, spurious I/O operations to conceal real traffic patterns. Such countermeasures are readily implemented but the performance impact of these countermeasures in most configurations is so great as to effectively preclude their adoption. Thus the only option is to select a configuration which exhibits an acceptable level of susceptibility to traffic analysis. This shortcoming with respect to traffic analysis is analogous to that presented by the encrypted bus approach, but here the level of traffic analysis detail available to an intruder is greater than in corresponding encrypted bus configurations, i.e., specific addresses are visible. This suggests that if traffic analysis is viewed as a serious problem, encrypted bus systems may be preferred over comparable encrypted storage configurations.

The encryption techniques employed for storage protection must conceal the data in the storage unit, provide a means for associating an ID with the unit, support detection of modification of the unit and distinguish among successive *versions* of the unit. This last point is very important and deserves further explanation. The IDs associated with storage units are generally reused, referring to different data over time. This is certainly true of the addresses used for primary and secondary memory IDs, except in the case of *write-once* media such as video-disks. For archival storage the problem arises if file names are used as IDs, unless the names are further qualified in some way, e.g., marked with the time and date of archival

unit creation. Most software is written under the (implicit) assumption that no malevolent entity will attempt to violate system integrity by taking advantage of ID reuse. To avoid this problem, IDs will be augmented, where necessary, with a *version tag* (VT) to provide *version differentiated IDs* that uniquely identify each distinct storage unit over time.

In order to fulfill the security requirements set forth in the preceding section, the following techniques are employed. First, each storage unit is encrypted using a cipher method employing an initialization vector formed from the unit's ID and VT. Encryption with an appropriate cipher method conceals patterns within a storage unit. The use of an IV based on the ID and the VT conceals patterns across unit boundaries and across versions of a unit. Second, associated with each storage unit is an error detection code (EDC)[11] calculated on the ID and VT as well as the data in the unit. This EDC detects modification of the data and, because it covers the ID and VT, it detects attempts to return other than the requested unit, i.e., a unit with the wrong ID or VT. Finally, a *version tag table* (VTT), keyed by storage unit ID, is maintained inside the TRM. This table provides a reference point for the timeliness guarantee by establishing the current VT associated with each storage unit. On each **Read**, the IV formed using the ID and the VT from the version tag table is employed to decipher the storage unit. If the storage unit is from the wrong location or is not the most recent one stored at the proper location, the storage unit will be improperly deciphered and the EDC check will fail.

Using these techniques, **Read** and **Write** operations are extended in the following fashion. On a **Write**, the VT for the storage unit is fetched from the VTT, updated and, with the ID, used as an IV in encrypting the unit before storing it outside the

---

[11]This EDC may be a conventional error detection code or it may be a cryptographic EDC (CEDC) or an authenticity/integrity check field (AICF) depending on the encryption mode employed.

TRM. The EDC is calculated on the ID, updated VT and the data, and it is encrypted and stored along with the unit. The updated VT is stored in the VTT, completing the operation. On a **Read**, the VT for the unit is fetched from the VTT and used with the ID as an IV for decrypting the unit as it is transferred into the TRM. The EDC is calculated on the ID, VT and the data as the transfer progresses and, when the transfer (data and EDC) is complete, the retrieved EDC is compared to the calculated EDC. If the EDC comparison succeeds, the storage unit is the one requested and it is intact, so processing can proceed securely in the TRM. If the comparison fails, either the unit was modified or the wrong unit was returned (incorrect ID or VT) and the unit is invalid, e.g., it may be viewed as having an unrecoverable error.

Just as the simple model of security requirements in section 4.1 does not fully capture the vagaries of T&A storage, this simple model of secure operation must be modified slightly to encompass **Read** operations for encrypted T&A storage. There is no need for a VTT for transfer units since these units are not created by the TRM and are not modified by the TRM. Instead, a version differentiated name is recorded with the transfer unit for use in decryption. Thus a **Read** of a transfer unit involves no fetch of a VTT entry. A VTT is required for archival storage to track the archival unit containing the most recent copy of each file with the most-recent-only reloading constraint. A table containing the IDs of all non-reloadable files also must be maintained. These tables perform the same functions as those described for the encrypted bus approach designs in sections 3.6.4 and 3.6.5. Since some files may be reloaded from other than the most recent archival unit copy (unconstrained reloading), the version differentiated name is recorded with each archival unit.

Finally, it is necessary to control DMA access to storage locations within the main TRM in the case of **SYSTEM E** and **SYSTEM F**. The individual **(write)** bus transactions that implement **Read** operations must be restricted to appropriate

primary memory locations, otherwise data in primary memory may be destroyed. This same problem arises in the encrypted bus approach and in the monolithic-TRM design in the context of aggregate transfers by non-secure DMA devices and the same solution is applied here. The secure storage interface (SSI) must act as a filter to restrict access to locations within the TRM. This applies not only to encrypted data transfers but also to accesses by non-secure DMA devices, just as in the encrypted bus approach. For each memory region that is accessible from outside the TRM, the SSI must be aware of the bounds of the region, whether **read** or **write** (or both) transactions are allowed and whether the transactions involve encrypted or cleartext data. Furthermore, the SSI must contain intra-TRM bus traffic, not repeating it onto the bus segment outside the TRM. This restriction is readily implemented by adopting the convention of assigning bus addresses that use a bit or two to distinguish between devices inside and outside of the TRM as described earlier.

The preceding discussion outlines the general techniques employed for securing encrypted storage at each level, but it does not describe all of the details involved. For example, it does not specify particular encryption techniques nor EDC computation strategies. Reliability measures and recovery strategies have not been discussed nor have the problems of storing large VTTs inside small TRMs. Tradeoffs in performance versus security related to the size of VTs and EDCs also must be addressed. The following sections deal with these problems, specifying the details of encrypted storage management for T&A storage, secondary storage and primary memory. Readers who do not wish to delve into these details should proceed to section 4.6 (page 208) for a summary of the highlights and the conclusions of this chapter.

4.3 Techniques for Encrypted Transfer and Archival Storage

The first issue to be resolved in filling in the details of secure T&A storage management is the selection of an encryption mode and an EDC calculation strategy. Transfer of an archival or a transfer unit between T&A storage and primary memory takes place at the speed of the T&A storage device, so the cipher method employed need not exhibit especially low delay, i.e., an extra cryptographic cycle or two on each unit transfer is acceptable. To avoid the need for additional hardware in the TRM for EDC or CEDC calculation (an EDC chip or an extra crypto chip) a cipher method with forward error propagation is employed. Since storage units at this level are relatively large (one or more files) and space is not at a premium, precise matching of encryption granularity and storage unit length is not a requirement. These observations suggest that block chaining with plaintext/ciphertext feedback (PCBC) is an appropriate cipher method for this application (see section 2.3). A predictable bit pattern embedded in the string at a known point serves as an authenticity/integrity check field (AICF) protecting all of the text preceding it. A version differentiated name employed as an IV is implicitly included in such an AICF.

4.3.1 Version Differentiated Names and the Archival Unit VTT

The next issue to be resolved is the form of version differentiated names for T&A storage and the related topic of a VTT for archival units. Clients and subsystem writers often think of T&A storage in terms of the names of the files recorded on the media. However, transfer and archival units may contain several files grouped to reflect logical dependencies among them, so individual file names are not always appropriate as IDs for these storage units. Moreover character string file names must be qualified in some way to distinguish successive archival units of the same

file (or groups of files). To avoid these problems, a unique bit-string identifier (a UID) is assigned as a version differentiated ID for each transfer or archival unit. Media used for transfer or archival storage usually contain a catalog that maps file names to their location(s) on the media and this catalog is easily expanded to provide a file-name-to-UID mapping. For archival units with the most-recent-only reloading constraint, a second map is needed: an *archival VTT* that associates with a file the UID of the most recent archival unit containing the file. (Non-reloadable files also are included in this table, using a distinguished UID to differentiate them.) The archival VTT is maintained on secondary storage as a table of file names and UIDs for files exhibiting this reloading constraint.

4.3.2 Format of Transfer and Archival Units

Figure 4-4 illustrates a sample format for an extended media catalog (containing storage unit UIDs) and for transfer and archival units (the two are quite similar). Note that the media catalog is unencrypted and is non-standard only in the addition of the UID field to each entry. However, each storage unit (transfer or archival) is encrypted. The unit begins with a header describing the unit and the files contained therein. The exact fields contained in the header will be system- and media-specific but should include the unit type (transfer or archival), header and total unit length, etc. Typical file entries would contain the file name, length, reloading constraint and other attributes included as an aid in (re)constructing secondary storage catalog entries. An AICF is appended to the header, providing a check on it, and the files follow this AICF directly. The entire unit, from header through final AICF, is encrypted as a continuous bit string using the PCBC cipher method noted above. In principle, only this final AICF is required but, since the header is used to control reloading, the header AICF is included to detect errors that might result in file system damage before the final AICF is encountered.

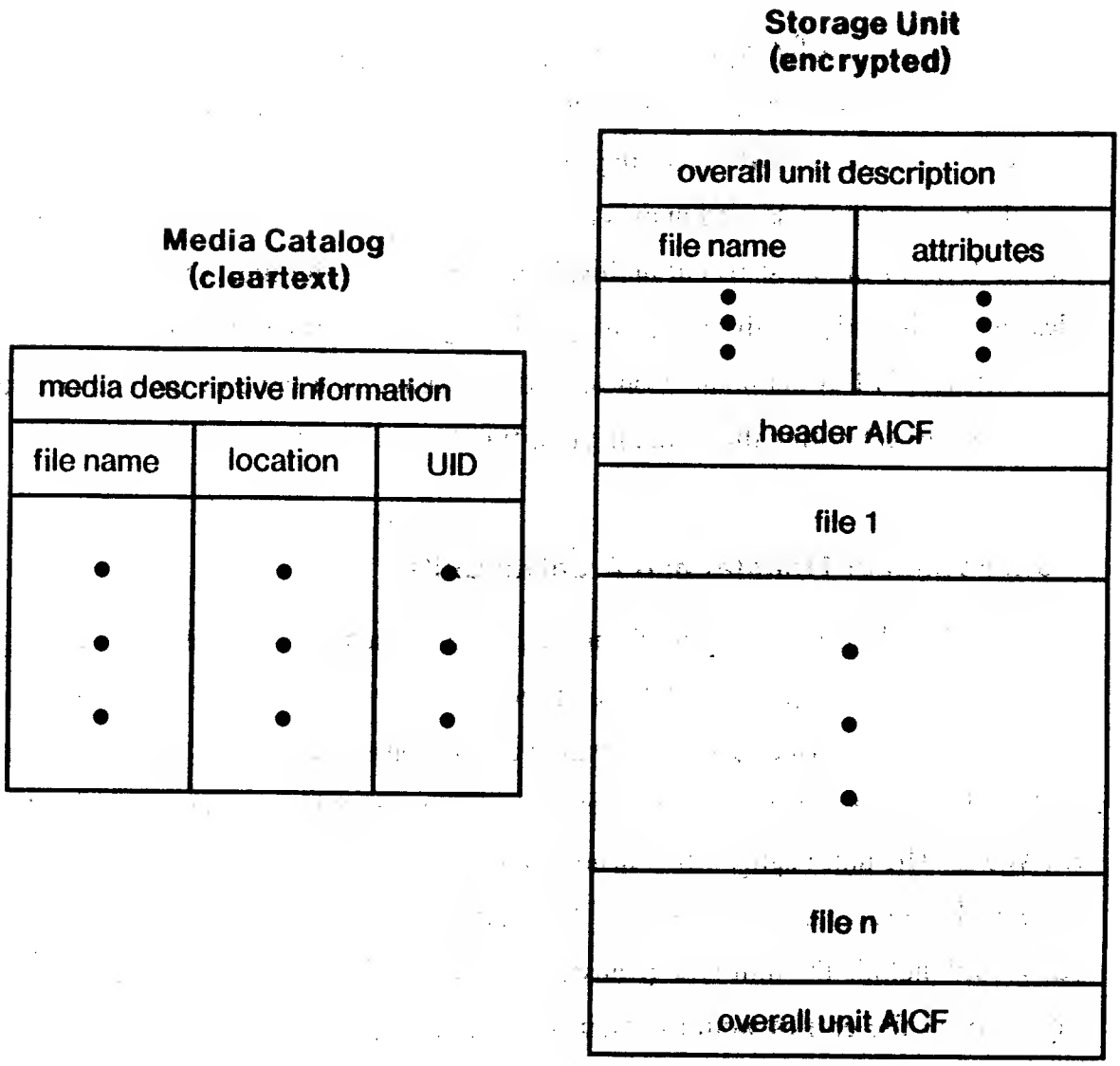


Figure 4-4: Format of Secure T&A Storage Media

Although the format of encrypted T&A media is similar for both transfer and archival purposes, there may be a difference in the key used to encipher the media. If transfer units are enciphered using the master key associated with a TRM, the units cannot be recorded until the target TRM is known. Demand recording of

transfer units is quite feasible for mail-order sales of proprietary software and could be carried out at local stores using high speed communication facilities to transmit the units for local recording. (Network-based distribution of external software is carried out in this approach just as it was described initially in section 2.3.4.) Alternatively, transfer units can be pre-recorded under randomly selected keys, which are then enciphered under the master key of the target TRM. This is essentially the same technique employed in the encrypted bus approach (for distribution of TRM components) and it requires only low speed communication between a local store and the vendor. In this approach the encrypted key can be recorded, at the local store, in a reserved location in the media catalog, making life somewhat more convenient for the client. The former distribution method is preferred since it means that the TRM need deal with only a single key for all encrypted storage, but the latter method can be employed if necessary.

4.3.3 I/O Operations on T&A Storage

It is now appropriate to examine the details of **Read** and **Write** operations on transfer and archival storage units. Remember that these storage units may consist of as little as a single file or may be a collection of a number of files. First, consider operations on transfer units. These units are **Read** by TRMs to initially load external software but TRMs are not allowed to **Write** these units. (The TRM operating system controls all encrypted I/O so it is capable of enforcing this prohibition.) To **Read** a transfer unit, the media containing the unit is mounted, the (cleartext) media catalog is scanned to determine the location and UID of the unit of interest (or of any file contained therein). This UID is loaded as an IV in an SSI crypto device in preparation for decrypting the transfer unit. (If transfer units on the T&A media are encrypted under a key other than the TRM master, then the encrypted form of this key is retrieved from the media catalog and loaded along with the UID.)

Next, the unit header is decrypted and transferred to primary memory where it is checked (using the embedded AICF and the header length constraint) and used to establish entries in the file system catalog for the files in the unit. Note that transfer units may serve as archival units for the programs and databases that constitute a protected subsystem, since the files on these units are non-modifiable, so file system entries may already exist for some of the file in the unit. If so, these entries are deleted when encountered in this phase of the unit **Read** operation, to ensure that the file system entries are consistent. However, any non-reloadable files contained in the transfer unit are not deleted if encountered. Rather a check is made against the archival VTT to ensure that any non-reloadable files in the transfer unit do not currently exist and have not existed previously (and were later destroyed). Non-reloadable files being loaded for the first time are recorded in the archival VTT to preclude any violation of this constraint. Each file in the unit is decrypted and transferred to primary memory and entered into the file system in secondary storage. When the last file has been transferred, the AICF covering the unit is checked. If this check succeeds, an *OK flag* in each file system entry just loaded is set to **TRUE**, indicating that the entire unit has been loaded successfully.

For archival units, both **Read** and **Write** operations are supported. An archival unit is created (a **Write**) by a call on the TRM operating system specifying the collection of files that are collected together to form the unit. External software invokes this operation on its mutable databases (or on the software itself) either periodically or when requested by the client. The operation begins with the mounting of archival media. The (unencrypted) media catalog is transferred to primary memory and modified to contain an entry for the new archival unit (virgin media is initialized with a null catalog). The unit header is constructed, gathering information from file system entries for each member of the unit, encrypted and transferred to the media. Then each file is encrypted as part of a continuous cryptographic chain and transferred to the media with an AICF appended to the end, and the updated media catalog is re-written.

Reloading an archival unit (a **Read**) is very similar to loading a transfer unit but the impetus is generally different. Usually the operation is triggered by damage to data in secondary memory, but it also may result from a program error or a client's decision to "roll-back the clock" with respect to some processing. A request to reload any file in an archival unit results in reloading all of the files in the unit (to ensure consistency). When reloading an archival unit, reloading constraints associated with the files in the unit must be checked. These constraints will be uniform for all files in the unit, i.e., all will either be most-recent-only or unconstrained. Only if the unit consists of most-recent-only files does the **Read** operation check the UID specified in the media catalog against the UID from the archival VTT and require that the two must match. Like the **Read** of a transfer unit, any files in the archival unit which already exist in the file system are deleted to ensure consistency. Thus a **Read** operation on an archival unit is almost identical to a **Read** operation performed on a transfer unit.

4.3.4 Robustness of the Archival Storage Protection Measures

If the archival VTT is damaged, files with the most-recent-only reloading constraint cannot be reloaded (since there is no way to determine which archival unit contains the most recent copy of the files). This type of damage need not preclude reloading of files that do not possess this constraint since the archival units for such files can be examined to determine their (lack of) reloading constraints. To enhance system robustness, the archival VTT should itself be archived (as a most-recent-only file), but this poses a problem. If the archival VTT is damaged and its most recent archival copy is reloaded, the entries for most-recent-only files archived since the archival VTT copy was created are lost, violating the most-recent-only constraint! To avoid this problem, updates to the archival VTT must be recorded in a non-reloadable file, the *archival VTT update file*, which is erased every time the

archival VTT is archived,. The UID of the current archival copy of the archival VTT must be maintained in some highly reliable fashion within the TRM, e.g., in non-volatile memory.

These measures allow recovery from a wide range of secondary storage failures affecting files and catalogs. Even file system catalogs can be archived (with the most-recent-only attribute) and reloaded to facilitate recovery from failures that damage these catalogs. In fact, these measures are so effective in promoting system robustness that they might create an opportunity to violate security provisions relating to non-reloadable files. A problem would arise if a non-reloadable file could be created, used and destroyed along with any record of its existence. To avoid this problem, when a file with the non-reloadable attribute is created, its file name is recorded in the archival VTT and is marked as a non-reloadable rather than a most-recent-only file (by using a distinguished value for a UID). Since updates to the archival VTT are protected by being recorded in the archival VTT update file until the archival VTT is archived, this solves the problem of *lost* non-reloadable files. When a subsystem attempts to create a non-reloadable file (or when a transfer unit containing a non-reloadable file is loaded), the file name is checked against the archival VTT to prevent violation of the timeliness guarantee, and an entry is created only if this is a new non-reloadable file.

This existence of the archival VTT does not enhance system robustness with respect to non-reloadable files (If such a file is damaged it is lost.), and it might even diminish robustness. If both the archival VTT and its update file are lost, no new non-reloadable files can be created or loaded from transfer storage and no most-recent-only file can be reloaded. However the loss of both of these files can be made very unlikely. The loss of any non-reloadable file is a very serious matter since it precludes use of the external software that employs the file. This suggests that non-reloadable files, including the archival VTT update file, should receive

special consideration from the file system. For example, such files can be recorded at two physical locations in secondary storage and have similarly redundant catalog entries to reduce the likelihood of their loss. Note that non-reloadable files are expected to constitute a relatively small fraction of all files, and may not occur at all in many systems, so these extraordinary robustness measures should not have a significant impact on the system.

4.3.5 Effects on Performance, Storage Utilization and the Operating System

Now that the description of protection measures for T&A storage is complete, it is appropriate to consider the effects of these measures on TRM operating system structure, system performance and storage utilization. The TRM operating system provides three new (or enhanced) functions: the **Read** operation for transfer units and the **Read** and **Write** operations for archival units. These operations have been described in some detail and are fairly simple. The operating system must make special provisions for creation and management of non-reloadable files, but some of these provisions would be required even in standard systems. System performance should not be significantly affected by the proposed measures; operations involving T&A storage are relatively infrequent, and the cryptographic transformations should not prove a bottleneck but only add a small delay to DMA transfers involving this storage. Delays will result from checking the archival VTT during reloading of most-recent-only files and creation or initial loading of non-reloadable files, but these are infrequent operations and thus the effect is not severe.

With respect to storage utilization, the protection measures increase the sizes of media catalogs and T&A storage units, and require two new files: the archival VTT and its update file. Catalogs for T&A media grow to accommodate storage unit UIDs whereas storage units grow to include reloading constraints and AICFs (and

may require padding for encryption). A 32-bit AICF should provide adequate protection for these storage units, especially since two such fields are contained in each unit. The UID associated with each unit should be large enough to identify every archival unit ever produced by a given TRM and to distinguish every distribution unit provided for a given TRM. A 32-bit UID permits a vendor to provide over 4 billion distribution units to a single TRM and supports archival unit creation at the rate of one per second for over 120 years. The IV used for encrypting/decrypting storage units should be a full 64 bits, so the 32-bit UID is augmented with 32 additional bits. Two of these additional 32 bits are used to distinguish among UIDs employed for archival, transfer and secondary storage units whereas the remaining 30 bits are unique per TRM. (This last set of bits may be viewed as an extension of the TRM master key.)

The increases in space on T&A media due to AICFs and UIDs are negligible (probably $<< 1\%$) since the storage units are files or groups of files. Some secondary storage space is devoted to the archival VTT and its update file, and the media containing these tables must be mounted for creation of non-reloadable files and reloading of most-recent-only files. Files with these reloading constraints are not expected to be the norm, so the archival VTT and its update file will not be too large. Thus the effects on storage utilization brought about by the measures are not expected to be significant. The impact on overall system robustness also should be minimal. The two new types of secondary storage data introduced to support encrypted archival storage, the archival VTT and its update file, are critical to system operation. However, the archival VTT is archivable and its update file is expected to be replicated in storage and catalog entries, like other non-reloadable files. Thus, only if both of these files are destroyed simultaneously will the system suffer irreparable damage.

4.4 Techniques for Secondary Storage

The protection measures presented in this section follow very closely the basic concepts presented in section 4.2. In this context, storage unit IDs are sector addresses qualified by the ID of the media containing the unit. The VTT, implicitly indexed by sector address, contains the VT associated with each sector for every encrypted secondary storage volume registered with the system. The integrity, authenticity and timeliness requirements are exactly as stated in section 4.1, with no exceptions. Thus **Read** and **Write** operations (sector transfers) proceed just as described in section 4.2. Even though performance degradation in storage unit transfers is more critical at this level than at the T&A level, the same cryptographic method is employed. Throughput with this method is more than adequate (even using a single crypto chip) and the added delay is still a negligible fraction (<< 1%) of total sector transfer time. A 32-bit AICF is appended to each sector, increasing sector size by about .75%.

4.4.1 The VTT Hierarchy

The major problem with this obvious approach is that it is impractical to maintain a secondary storage VTT within the TRM boundary. For example, a typical 30M-byte (unformatted) disk contains about 50,000 512-byte sectors. If each VTT entry consists of a 32-bit VT (assume the address of the sector being protected is implied by index of the VT in the VTT), the resulting VTT occupies 200,000 bytes and this covers only a single volume! The amount of secondary storage devoted to the secondary storage VTT is not a concern, but it is generally impractical to maintain this VTT inside a TRM. This space problem suggests that the secondary storage VTT should be hierarchically organized, with only the root maintained within the TRM. Figure 4-5 illustrates a 4-level hierarchy for the secondary storage VTT.

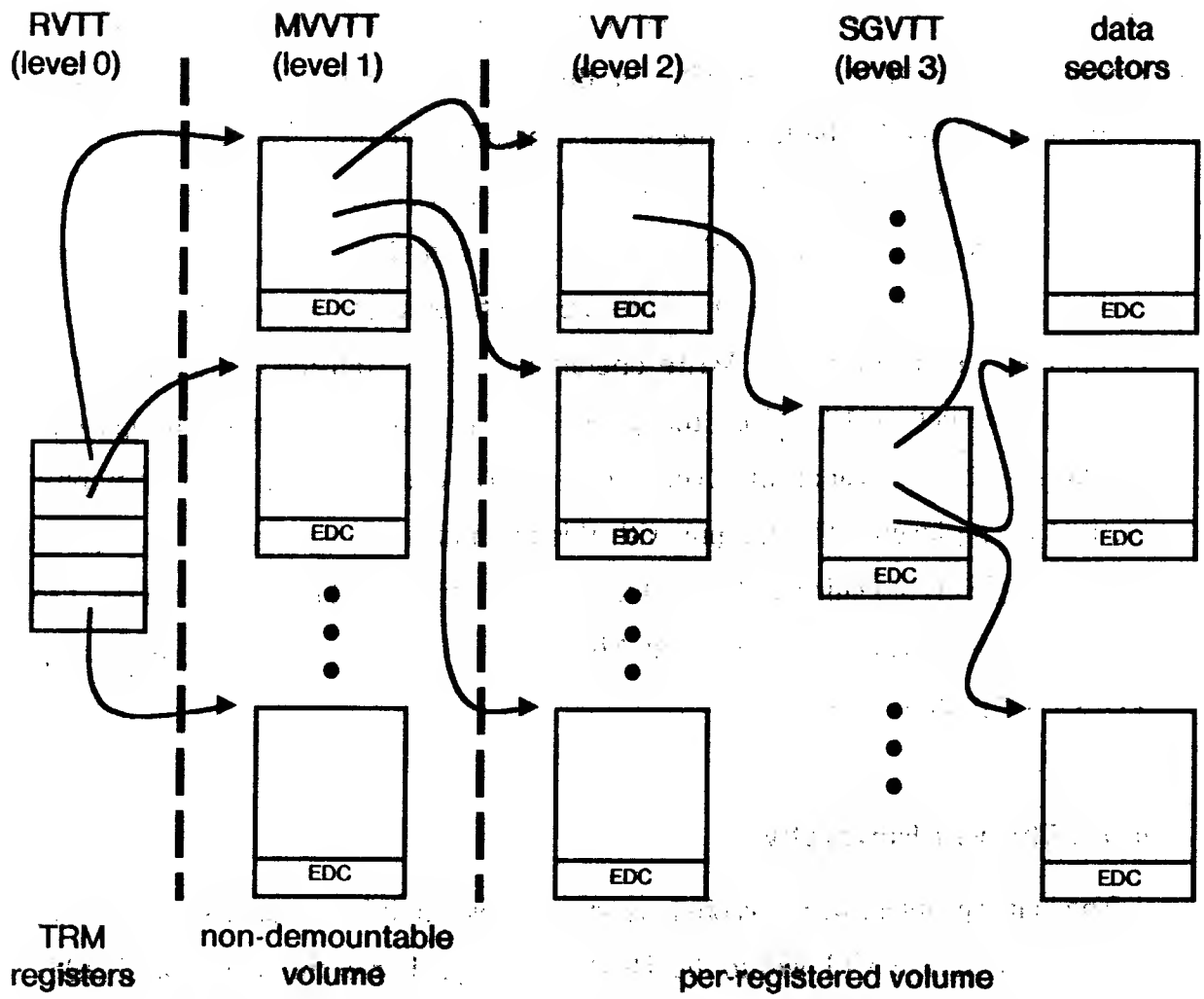


Figure 4-5: Hierarchic Organization of Secondary Storage VTT

In this figure, the arrows indicate which sectors are covered by VTT entries in a given level of the VTT hierarchy. Below the root VTT (RVTT) (level 0) is the master volume VTT (MVVTT) (level 1) which contains one entry for each encrypted volume registered with the system. Each volume contains a volume VTT (VVTT) (level 2) and below it is the sector group VTT (SGVTT) (level 3). At each level of the hierarchy a VTT protects the sectors at the next level with the bottom

level (sector group) VTT protecting data sectors. This recursive structure protects every sector in secondary storage in the same fashion by using the associated AICF and the corresponding VT recorded in the preceding level of the hierarchy; hence there is no difference in the protection afforded a data sector versus a VTT sector at any level.

The root VTT contains the volume ID and addresses of each sector occupied by the master volume VTT as well as a VT for each of these sectors, all maintained in non-volatile storage within the TRM. Each master volume VTT entry contains the ID of the volume represented, the addresses and VTs for the sectors that make up the volume VTT and other supporting information. At the volume VTT and sector group VTT level the addresses of the sectors being protected need not be explicitly stored along with the VTs, but can be implicitly derivable from the index of the VTs in the VTTs. Implicit addressing in the volume VTT entries requires the sector group VTT sectors to be contiguous or to be dispersed about the volume in some fixed pattern (to optimize seek time). The sector group VTT always employs implicit addressing since it is usually trivial to arrange for the sectors covered by these entries to be contiguous. Throughout this chapter the assumption is made that the sector group VTT sectors are contiguous in order to reduce the amount of space devoted to volume VTT entries. (This assumption does not affect the security of the design.)

This hierarchic structure avoids the need to store the entire VTT inside the TRM, but it transforms each reference to secondary storage into a chain of references through the levels of the hierarchy, as shown in Figure 4-5. Consider a reference to a sector with ID (fully qualified address) vx, where v is the volume ID, and x is a sector address. The reference chain begins at the root VTT with the volume ID and addresses of the master volume VTT and the VTs for each master volume VTT sector. Using this information from the root VTT, the master volume VTT sector

containing the entry for volume v is fetched. (It may be necessary to serially search this table if volume IDs are sparse or if entries in the master volume VTT are of variable size.) The VT and the address of the appropriate sector of the volume VTT is selected from this master volume VTT entry by examining the target address x. This volume VTT sector is fetched and the VT and address of the appropriate sector of the sector group VTT is selected in the same fashion. Finally this sector group VTT sector is fetched and the VT for the target sector is selected.

Following this chain of references results in at least 4 sector fetches (perhaps more depending on the master volume VTT organization) as compared to the single fetch required in a standard system. This sort of problem commonly arises in hierarchic address translation and it is usually solved by encaching portions of the translation tables to short circuit the reference chain. In this context encaching means keeping portions of the master volume VTT, volume VTT and sector group VTT in primary memory to reduce extra sector fetches. From the master volume VTT, entries that correspond to currently *mounted* volumes should be cached. Since the systems of interest are small and master volume VTT entries are small (about 64-256 bytes depending on the capacity of the volume), these entries (perhaps 2-5) occupy a negligible percentage ($<<1\%$) of primary memory. At the volume VTT level the amount of information to be cached depends on the size and number of mounted volumes and the size of primary memory. For example, small and medium size volumes, e.g., 4M-byte floppy disks through 30M-byte fixed disks, have volume VTTs that occupy about 1-4 sectors, so it is probably feasible to cache the entire volume VTT for such volumes. However, for large volumes, e.g., 300M-byte demountable disks, the volume VTT is very large, about 36 sectors, making it likely that only portions of this table will be cached at any point in time.

Proceeding to the bottom of the hierarchy, sector group VTTs will range in size from about 64 sectors for a small disk to about 500 for a medium size disk and up to

4000 for a large disk. Thus it is usually infeasible to cache the entire sector group VTT of a volume in primary memory. In fact, it is often inappropriate to cache whole sector group VTT sectors since, in the worst case (if each sector in primary memory comes from a location not covered by any other sector group VTT sector in the cache), there must be one sector group VTT cache entry for each sector in primary memory. This worst case behavior could result in the sector group VTT cache occupying 50% of primary memory and thus motivates caching only portions of sector group VTT sectors, e.g., 8 word pieces instead of full 128-word sectors. In this fashion only about 8% of primary memory is required to cope with even the worst case scenario for the sector group VTT cache. Overall, the caches for the master volume VTT, volume VTT and sector group VTT may occupy about 10% of primary memory if organized in this fashion.

4.4.2 I/O Operations on Secondary Storage

Using this VTT hierarchy, **Read** and **Write** operations proceed as follows. On a **Read**, the volume ID and sector address are combined with the sector VT to form an IV for decrypting the target sector. When the sector has been decrypted, the AICF following it is checked against the computed value and the operation is aborted only if the check fails. On a **Write**, the VT for the sector is fetched from its cache, updated and used as above to form an IV for encrypting the sector and the trailing AICF. When the **Write** completes, the VT cache entry is updated and, at some later time, the VTT in secondary storage is updated. These descriptions apply to operations on all sectors and the VTT updates propagate up through the hierarchy. When a volume is mounted, the master volume VTT is **Read** and searched for the entry for the mounted volume, then this entry is stored in the master volume VTT cache. If the entire volume VTT of the volume is cached, it is **Read**, otherwise sectors (or sub-sector portions) of the volume VTT are **Read** as needed.

References to data sectors proceed as noted above if there is a hit on the sector group VTT cache. A miss on this cache results in flushing a cache entry, if none are available, and the appropriate sector group VTT sector is **Read**, using the volume VTT cache for the **Read** of the sector group VTT. If a modified sector group VTT cache entry is flushed, it must be written back. This entails a **Read** of the containing sector group VTT sector, an update of the sector (which is noted in the VOLUME VTT cache), and a **Write** of the sector. A miss on the volume VTT cache is handled analogously, but will be simpler if volume VTT cache entries are whole sectors rather than sub-sector pieces. Periodically, or when requested by the client or external software, all modified entries in the VTT caches can be flushed, starting at level 4 and proceeding through an update of the root in the TRM, producing a non-volatile, consistent version of the VTT hierarchy in secondary storage. Until this flushing operation takes place, changes to files (in particular, modifications to non-reloadable files), are not permanently recorded in the VTTs and thus may be undetectably *undone* by an intruder.

This VTT hierarchy is organized solely around the physical media without regard to file system structure, thus demonstrating that these techniques can be employed independently of such structure. However, it may be advantageous to integrate the hierarchy with the file system structure. For example, the sector group VTT VTs can be integrated with the tables used to map sectors of a file to their secondary storage locations, and the volume VTT can be extended to cover these integrated file maps/VTTs. The file maps will grow by about 200% (due to the presence of VTs) but since the cache space devoted to such maps is often on the order of 1.5-2.5% of primary memory, the cached level 3 VTs will require only 3-5% instead of the 8% of primary memory noted above. Integrating the sector group VTT and file map caches takes advantage of the logical locality of reference implicit in file structure. In this way, whenever a sector can be directly referenced, by virtue of its file map being in the cache, its VT also is present, improving the sector group VTT cache hit

rate and simplifying the lookup procedure for sector group VTT entries! The only drawback to this approach is that the volume VTT becomes larger (about 50%) since it covers more data (file maps as well as level 3 VTs), and thus the volume VTT cache grows or its percentage coverage decreases.

4.4.3 Performance, Robustness and Storage Utilization Issues

It is now appropriate to evaluate the impact of these secondary storage protection measures on robustness, storage utilization and performance. In secondary storage five types of sectors are distinguishable with respect to their impact on system robustness: reloadable files and catalogs, non-reloadable files (including the archival VTT update file) and their catalog entries, sector group VTTs, volume VTTs and the master volume VTT. The first type is present in all systems, the next arises from encrypted archival storage security measures and the last three support encrypted secondary storage. Thus the question is how damage to the last three type of sectors affects the other sector types, in particular how it affects non-reloadable files. A reasonable goal is to prevent the loss of any single sector from causing an irrecoverable loss of data, i.e., loss of a non-reloadable file or its catalog entries. Damage to a sector group VTT sector results in loss of the 128 sectors covered by it. This may include ordinary files, catalogs and non-reloadable files. To reduce the likelihood of losing a non-reloadable file, the replicated non-reloadable file sectors and catalog entries should be covered by different sector group VTT sectors. Integration of the level 3 VTs with file maps makes this easier because of the relationship between files and level 3 VT sectors.

Damage to a volume VTT sector results in the loss of 128 sectors of sector group VTT, or of file maps and level 3 VTs, and, transitively, of 16,384 file and catalog sectors. This is a significant loss of information and makes it difficult to guarantee that the replicated copies of a non-reloadable file and its catalog entries are not

covered by a single volume VTT sector. Since only a few sectors (1-64) are devoted to a volume VTT on each volume and since I/O on these sectors is relatively infrequent, it is feasible to replicate these sectors on each volume. A similar argument applies to the master volume VTT, which is both smaller and more important in its coverage. This replication requires slightly larger master volume VTT entries (to contain the addresses of both volume VTTs on each volume) and more non-volatile memory in the TRM (for the dual master volume VTT addresses), but these are very small increases in storage utilization. These added precautions yield a secondary storage system in which no single sector failure can result in an irrecoverable loss of data.

These protection measures have only a very slight effect on secondary storage utilization. Together, the space occupied by each sector group VTT (or its integrated file map alternative), volume VTT (including backup copy) and the per sector AICFs amounts to about 2% of a formatted volume. The space devoted to the master volume VTT and its backup copy should constitute a negligible fraction (<<1%) of the storage on a permanently mounted volume. The caches for level 3 VTs require about 3-5% of primary memory if the VTs are integrated with file maps. The percentage of primary memory devoted to the volume VTT cache depends on the size of memory, the capacity and number of mounted secondary storage volumes and the fraction of each volume VTT required in the cache for acceptable performance. For example, the volume VTTs for two 30M-byte disks occupy about 2% of a 256K-byte primary memory. Thus a total of about 4-7% of primary memory may be dedicated to VTT caches. (The master volume VTT cache is a negligible contributor to this total.)

System performance is affected in several ways by the secondary storage protection measures. On each **Read** of a file or catalog, there is a delay resulting from the transactions required to control the secure storage interface (SSI), to fetch

the AICF word and to decrypt the last two data words in the sector. Controlling the SSI involves loading the sector address, volume ID and VT to form the IV, and loading the primary memory sector frame address and access mode (read or write) to restrict DMA access. The bus transactions required to control the SSI can be carried out during the accessing of the secondary storage device before the data arrives, given the average access time of secondary storage devices. Thus these transactions do not contribute to delay, they only increase bus utilization slightly. Moreover, the decryption of the last two data words can be overlapped with the fetch of the AICF word so the total delay experienced is the maximum of these two operations. For unbuffered secondary storage devices, the AICF transfer requires greater time, but it is only about 3μs for a 10 M-bit/second transfer rate, a negligible (<<1%) increase in total **Read** time.

If level 3 VTs are not integrated with file maps, misses can occur on the sector group VTT cache, resulting in significant delays. Such a miss requires locating a cache entry to flush, updating the secondary storage sector group VTT sector if this cache entry has been modified (this requires a **Read** and a **Write** on the relevant sector group VTT sector) and performing a **Read** on the sector group VTT sector containing the required VT. Thus either 1 or 3 extra secondary storage operations are required on a miss and this could noticeably degrade performance if the cache did not achieve a high hit rate. For example, a 90% hit rate might result in a 20% delay on secondary storage I/O and a 95% hit rate yields a 10% delay. This strongly motivates the integration of level 3 VTs and file maps, since such integration eliminates VT cache misses at this level. (The only way a file can be referenced is if its map is in primary memory.)

Employing this integration strategy, cache misses at the volume VTT level occur at the point when file maps are **Read**. For many small and medium capacity volumes, the entire volume VTT can be cached, completely avoiding misses at this

level. Even if caching of whole volume VTTs is impractical, the volume VTT cache should accommodate a very large percentage of the volume VTT, achieving a very high hit rate and minimizing the delays due to misses. Only in the case of large volumes is there likely to be any significant delay due to volume VTT cache misses. This suggests that very large volumes may best be handled by dividing them into multiple *virtual volumes* like the mini-disks employed by VM/370. The time required to fetch the master volume VTT entry for a volume when it is mounted is easily absorbed in the manual mounting process. It is very difficult to estimate the performance impact of the additional secondary storage I/O required when a VTT flush operation is undertaken, especially since the frequency of such operations is application- dependent. However it seems reasonable to assume that such operations are not so frequent as to significantly affect performance.

In the interest of improved performance and enhanced robustness, some bubble memory storage can be included within the TRM. The entire master volume VTT and the archival VTT update file can reside in this storage, eliminating the need for a permanently mounted volume containing these tables. Moreover, the complete volume VTTs and sector group VTTs for several mounted volumes can be cached in such storage. This would eliminate secondary storage transfers related to VTT management except when a volume is initially mounted and before it is demounted. Bubble memory access time is fast enough to fetch level 3 VTs from this cache instead of from primary memory (for non-bubble memory secondary storage devices). This configuration option is in no way essential to the design presented above, but the availability of high density (4 M-bit) bubble memory chips makes it a feasible means of enhancing system performance and reliability.

4.4.4 A Note on the Size of Secondary Storage VTs

Throughout this section the VTs have been described as 32-bit quantities. This distinguishes about 4.3 billion versions of a sector. For a data or catalog sector, a *maximum* rate for write-backs is probably on the order of 1 every 10 ms for a disk (assuming a transfer rate of about 10M bits/s, an average latency of about 9 ms and some system overhead). At this rate the VT of a single sector could be exhausted (wrap around) in about 1.36 years of continuous write-backs of that one sector. This rate of use is obviously much greater than would be expected in normal operation, perhaps by an order of magnitude, yet it is difficult to estimate a reasonable write-back rate. Thus some provision should be made to accommodate the possibility that a VT will be exhausted in the lifetime of a secondary storage volume. The method should provide for an orderly transition that allows the data recorded on the volume to be used as though nothing special had happened.

The proposed method involves two additions to master volume VTT entries and a new value to be held in non-volatile memory in the TRM. The master volume VTT additions consist of a field to track the maximum value attained by any (data sector) VT on the volume and another field to provide a volume UID used only for cryptographic purposes. The new value held in the TRM is a global counter used to generate these volume UIDs. The UIDs are used in forming the IVs employed in cryptographically transforming sectors on the volume, instead of simply using the logical volume ID described earlier. When a new volume is registered with the system the global counter noted above is incremented to generate a UID for that volume. When a threshold is reached on the per-volume, maximum VT value (indicating that a VT on the volume may soon be exhausted), the global counter is again incremented and the client is notified that the volume must be copied to a new volume. This new volume will be assigned the same logical volume ID used for addressing, but it will have a different volume UID. (The old volume later can be recycled into a new volume using this procedure.)

In copying the old volume to the new volume, each sector is re-encrypted using the IV formed from the new volume UID, the sector address, and a re-initialized sector VT. The volume UID field in the master volume VTT entry for the new volume is updated after the copy operation is complete and has been checked. The 64-bit IV used throughout this chapter is divided into four fields here. Two bits are used to distinguish among the four storage unit types: transfer units, archival units, sectors and cache lines (see section 4.3.5). Twenty bits are devoted to the sector address (allowing up to 1M sectors on a single volume) and 32 bits are devoted to the sector version tag. This leaves 12 bits for the volume UID, supporting over 2K volume versions over the lifetime of the system. Since it was noted above that it would take about a year to exhaust the sector VTs for a single volume at a maximum rate, this should prove to be an adequate number of volume versions!

4.5 Techniques for Encrypted Primary Memory

The protection measures developed for encrypted primary memory are similar, in many respects, to those described in section 4.4 for secondary storage. The integrity, authenticity and timeliness constraints for encrypted primary memory are exactly those stated in section 4.1 and imposed at the secondary storage level. In primary memory the storage units are cache lines and the IDs are the primary memory addresses of these lines. (It will become clear in this section why individual words are too small to be treated as storage units at this level.) Using the model developed in section 4.2, modifications to a storage unit are effected by a **Write** of the entire unit. Thus only write-back caches are applicable here, since write-through caches effect modifications through partial updates of cache lines. When a storage unit is transferred from T&A storage to secondary storage, it is transformed from the T&A representation to the secondary storage representation. The transfer or archival storage units is decrypted, its AICF is checked, it is divided into sectors and re-

encrypted with an AICF for each sector, and the relevant secondary storage VTT entries are updated. The inverse of this transformation takes place when files are archived.

Analogous procedures take place when an encrypted sector from secondary storage is transferred to primary memory and transformed into encrypted cache lines or vice versa. Configurations such as **SYSTEM H** provide a natural point, the bus coupler, for performing these transformations, whereas configurations such as **SYSTEM G** are unsuitable since they provide unmediated access (by DMA devices) to primary memory. Adopting the former configuration, there are two secure storage interfaces (SSIs) in the TRM: one interfacing to the I/O bus and the other to the memory bus. The I/O bus SSI controls **Read** and **Write** operations on T&A and secondary storage units and restricts access to primary memory by devices on that bus, whereas the memory bus SSI manages these operations for primary memory. For reasons of design simplicity, all data in primary memory is encrypted, including data stored and fetched by non-secure DMA devices under the control of the I/O bus SSI.

The VTT for encrypted primary memory is implicitly addressed by ID and it contains one entry for each cache line in primary memory. Since, in configurations such as **SYSTEM H**, there is essentially no storage within a TRM, a hierarchic VTT structure and VTT caching may be appropriate here, too. Despite these many similarities to encrypted secondary storage, there are several aspects of encrypted primary memory that distinguish it and which warrant special consideration. For example, storage units (cache lines) are so small that the space devoted to VTs and AICFs constitutes a significant fraction of the storage at this level. Special efforts are required to reduce this overhead to acceptable levels. Also transfers of cache lines across the TRM boundary (through the memory bus SSI) must take place at very high speeds and deliver the requested data with minimal additional delay. To

meet these stringent performance constraints, special care is required in the selection of cryptographic techniques for concealment and detection of modification. The following sections address these problems in describing encrypted primary memory techniques in detail.

4.5.1 Downsizing and Storage of EDCs

The EDCs (AICFs) and VTs employed for T&A and secondary storage are 32-bit fields. (Throughout this section and the next the term *EDC* will be used generically, encompassing AICFs and CEDCs as well as conventional EDCs.) The space devoted to EDCs, VTs and various auxiliary data structures, e.g., T&A storage unit headers, amount to less than 2% of the space occupied by the storage units being protected (even less for most T&A units). Cache lines for the systems of interest are only 16 or 32 bytes long, so 32-bit EDCs and VTs would require primary memory to grow by 25-50% to accommodate these fields! Although the per-bit cost of memory is declining rapidly, the storage overhead for VTs and EDCs would unacceptably increase system cost in most cases. This overhead can be reduced only through the use of smaller fields for the EDC and VT, e.g., cutting these fields in half. (The alternative of larger cache lines is rejected since the proposed 32-byte cache lines are already quite large for these small systems.) In the encrypted bus context it was suggested that a 16-bit EDC might be adequate for most applications and the same argument can be applied here. With such a small EDC, it is necessary to limit automatic retries when an error is encountered and to establish an error threshold which, if reached, causes the system to shut down and requires intervention by the vendor, as proposed in section 3.6.3.

It may appear that the adoption of a 16-bit (halfword) EDC for cache lines engenders a drastic response to errors but this response is justifiable. Note that this EDC does not replace the error detection and correction code usually employed

with solid-state memories, so only errors that evade that code will be dealt with by this security mechanism. This suggests that errors detected by this security are likely to be the result of tampering attempts and thus warrant a severe response. With an appropriate choice of error threshold it is unlikely that a non-malicious client will ever encounter this response. Since encrypted primary memory, like an encrypted bus, provides only a temporary repository for data, halting and restarting the system in the event of an error should not result in a significant loss of data.

One other aspect of EDC management for encrypted primary memory deserves mention: the location of EDCs. The mapping of cache lines to primary memory locations is very simple because the length of lines is normally an integral power of two. Any effort to append halfword EDCs to lines would require either a much more complex mapping or some form of non-standard primary memory interface, e.g., one in which the EDCs were implicitly addressed (and do not occupy a portion of the "normal" primary memory address space). Since one of the motivations for configuring systems of this sort is the ability to use "off-the-shelf" primary memory, this seems like a bad approach. The alternative is to group all the EDCs into a contiguous table in primary memory and to fetch the appropriate EDC using a separate bus transaction. This approach generates somewhat more bus traffic and delays delivery of the EDC, but in a cache-equipped system the additional bus traffic is not a major concern and the increased delay is not important due to other timing constraints (see section 4.5.4). Thus EDCs will be collected together in a table in primary memory.

4.5.2 Downsizing of VTs: The Cryptographic Refresh Process

Reducing the size of VTs is a more complex task. The VT must not be allowed to wraparound under a single key lest security weaknesses result (see section 2.3). The VT for a cache line is updated whenever a cache miss occurs that results in the

eviction of a modified instance of that line (a *dirty miss*). The worst case scenario for VT updates proceeds as follows. A modified cache line is evicted (a dirty miss); then a *clean miss* occurs (no write-back) on the line just evicted and, finally, a dirty miss occurs that evicts the line in question. This series of activities provides the minimum time between updates to the VT associated with a single cache line. A 32-bit VT would wraparound in several hours under this worst case scenario and for a 16-bit VT the time to cycle would be less than half a second, based on the operation timing figures developed in section 4.5.4. Of course this worst case scenario generates dirty misses on a single line much more frequently than one would expect to encounter in practice, but the very short wraparound time for a 16-bit VT poses a serious problem even for normal operational environments.

To avoid this problem, it is necessary to change the key used to encipher cache lines, before a VT can wrap around, since no weakness results if the duplicate VTs arise under different keys. Since there are 2^{56} distinct keys for the DES, there is no concern over running out of keys based on any practically attainable rate of key change. Thus one key, the TRM master key, is used to protect secondary storage units and, in some systems, T&A storage units, but a succession of random keys will be used to protect cache lines. The transition from one cache line key to the next must be carried out in a fashion that does not disrupt system operation nor degrade performance. The mechanism developed for this task can be thought of as a continuous *cryptographic refresh* of primary memory.

Cryptographic refresh is an activity (independent from the calculations taking place at the processor) directed by some control logic included in the memory bus SSI. It uses the crypto chips in this SSI along with some additional registers and a cache line buffer. Two working keys are identified in this SSI: WK_1 and WK_2. Before the cryptographic refresh process starts, all cache lines in primary memory are encrypted under WK_1. The process begins with the generation of a (pseudo)

random value for WK_2. A register, which tracks the progress of the process, is set to the address of the highest numbered cache line in primary memory. The VT for this line is retrieved from the VTT, the line is fetched from primary memory and decrypted and its EDC is fetched, decrypted and checked. Assuming no error is detected, the line is encrypted under the next working key (using a VT of 1) and stored in primary memory, the EDC is encrypted and stored, and the VTT is updated to reflect the reset VT. This process continues through all of primary memory until every cache line has been transformed, completing a pass of the refresh. Then WK_1 is set to WK_2 and the process begins again.

At any time during this process, it is possible to determine which of the two keys held in the crypto chips should be used to encipher/decipher a cache line by referring to the register that tracks the progress of the refresh pass. If the requested cache line is the one currently being processed, it is already buffered in the SSI (in the clear), so it is immediately available and the question of which key to use is avoided. This refresh process operates at the lowest priority with respect to use of the crypto chips and the memory bus, pre-empted by memory requests from the processor or from the I/O bus, thus it should not perceptibly affect system performance. The critical timing requirement for this process is that a refresh pass must complete before VT wraparound occurs. Equation 4-1 expresses the relationship between the mean time between cache write-backs (*MTBWB*) for a single line, the time required to refresh a cache line (*T*) and the amount of primary memory (*P*), expressed in cache lines, that can be refreshed before a 16-bit VT wraparound occurs. (The .9 factor arises from the observation that the memory bus and its SSI are idle, and thus available to the refresh process, about 90% of the time in systems configured in this fashion.)

$$P/T < .9 * 2^{16} * MTBWB \qquad (4\text{-}1)$$

The refresh of a cache line involves a **Read** of the line followed by a **Write** of the refreshed line, requiring about the same time as a dirty cache miss. In the worst case VT update scenario, the VT of a single line can be updated in about 1.5 times the dirty miss time (T = 1.5 * MTBWB) due to the inclusion of the clean miss between the two dirty misses. At this rate the maximum primary memory size would be a little over 2.3 Mbytes for 32-byte cache lines. However, as noted earlier, this especially abusive pattern of memory references is not likely to arise in practice and larger primary memory configurations can be supported if a mechanism is provided to prevent wraparound in the case of an attack based on maximum rate VT updating. To prevent a security breach, the memory bus SSI will refuse to write-back a cache line if its VT would wrap around (simple overflow detection), halting the system instead. Hence, in practice, very large primary memory configurations will be supported comfortably since the *MTBWB* is likely to be much longer than the worst case figure projected above. Thus the cryptographic refresh technique permits the use of small (16-bit) VTs without sacrificing security or degrading performance.

4.5.3 A VTT Hierarchy and VTT Cache Management

Employing 16-bit VTs, the cache line VTT requires 6.25% of the space devoted to cache lines, e.g., a 1M-byte primary memory needs a 65538-byte VTT. This VTT either can be contained wholly within the TRM or it can be hierarchically organized and stored in primary memory with only a portion of it cached within the TRM. Although this choice is analogous to that presented at the level of encrypted secondary storage, there are some important differences. For example, if the VTT is TRM-resident, it probably will be stored using primary memory chips since high speed (cache) memory chips offer only a slight overall performance advantage. But if a VTT cache is employed, the higher speed chips may be required in the TRM to

offset the added delays imposed by the cache lookup procedure. Moreover, the quantity of primary memory that is attached to a system is often more tightly bounded than the number of secondary storage volumes that may be registered with a system, making it feasible to construct a TRM with a VTT large enough to cover a likely range of primary memory configurations. Finally, the complexity of the control logic and the size of the auxiliary storage needed for the management of the VTT cache also motivate incorporation of the whole VTT in the TRM. To understand the tradeoffs involved, it is necessary to examine the details of managing a hierarchic VTT and its cache versus a TRM-resident VTT.

The organization and management of a TRM-resident VTT is trivial. Storage is provided so that each cache line in primary memory has a corresponding 16-bit VT, indexed implicitly by the cache line address. A lookup of a VT is accomplished in one access to this table and should require about two cycles: one cycle for memory access and one cycle for (round-trip) transport within the TRM. A store into the VTT of an updated VT is accomplished similarly and in the same amount of time. The cryptographic refresh process interacts smoothly with this arrangement. The disadvantages of this scheme are the increase in TRM size and complexity due to the inclusion of the memory chips for the VTT and the constraint placed on main memory configurations by the size of this VTT. If 64K-bit memory chips are employed, then a set of 9 (parity included) will support up to a 1M-byte primary memory. If 256K-bit chips are employed then a similar chip set will support up to a 4M-byte primary memory configuration.

If the VTT is not wholly TRM-resident, a simple, two-level hierarchy will be employed as part of a VTT encachement scheme. The bottom level of the hierarchy consists of the VTT divided into cache line-sized pieces and the top level (root) consists of VTs for these VTT lines. The *VTT root table* is permanently resident in the TRM along with the *VTT cache* and the *VTT cache lookup table.* This last table

is used to determine if the VT for a requested cache line is in the VTT cache and, if so, to locate that VT. Each VT in the VTT root table covers a cache line of VTs which in turn covers 16 data cache lines, so the VTT root occupies space equal to .2% of primary memory. The VTT cache contains one line for every line in the data cache, to accommodate a worst case situation in which each line in the data cache is covered by a different VTT cache line, plus a couple of additional entries for reasons explained later. (Note that entries in the VTT cache do not correspond directly to lines in the data cache since one VTT cache entry could cover up to 16 lines in the data cache.) Entries in the VTT cache are 32-byte lines, plus a *modified bit*, an *in-use bit* and a reference count for use by the replacement algorithm. This the VTT cache is roughly the same size as the data cache (about 3% larger).

The VTT cache lookup table contains one entry for each block of 16 data cache lines in primary memory, i.e, the set of data lines covered by a VTT line. If the VT for a data cache line is in the VTT cache, the corresponding lookup table entry contains the index of the containing VTT cache line, otherwise the entry is marked as empty. This table is about half the size of the VTT root table since the unit of coverage is the same and the VTT cache indices are about half the size of VTs. A likely size for the data cache is 8 Kbytes. Using 32-byte lines, a total of 256 lines fit in this cache, yielding a cache index size (for VTT cache lookup table entries) of 8 bits and a reference count (for VTT cache entries) of 8 bits. Thus, in total, the tables employed in the VTT caching scheme amount to about .4% of primary memory for the VTT root table and the VTT cache lookup table, and about 103% of the data cache for the VTT cache. For example, a 1M-byte primary memory system requires a total of about 12 Kbytes of additional storage within the TRM to hold the various tables and the VTT cache, compared to the 64-Kbyte VTT that would migrate into the TRM if caching were not employed. For a 2-Mbyte system, the figures are about 16K bytes versus 128 Kbytes.

The VTT cache operates as follows. When a (clean) data cache miss occurs, the VT for the requested cache line must be retrieved in order to decrypt this line. The VTT cache lookup table is checked to see if the required VT is present in the VTT cache. If the VT is present, the lookup table entry and the low order bits of the address of the requested cache line are used to index into the VTT cache. There the required VT is retrieved and the reference count for that VTT cache line is incremented. If the data cache miss was dirty (implying a write-back), the same procedure is followed so that the requested data line can be **Read** first, then the VT for the evicted line is retrieved as above, the *reference count* of the containing VTT cache line is decremented and the *modified* bit is set. (The VT for the evicted line is always present in the VT cache.) If the VT for the requested data line is not present, a VTT cache miss occurs. This miss must be processed before the data cache miss. Processing of a VTT cache miss is the same as for a data cache miss with the exception of the replacement mechanism.

The *reference count* associated with each VTT cache line reflects the number of data cache lines covered by it, and the in-use bit indicates if the entry is empty or occupied. Scanning of the VTT cache to free lines can take place either on a demand basis (when a VTT cache miss occurs) or as a background activity like cryptographic refresh. Lines in the VTT cache with a reference count of zero are eligible for replacement and, if unmodified, are marked as empty and ready for immediate reuse. Modified lines with a zero reference count are evicted, updating the VT entry in the root table, and then marked as empty. The two *extra* lines in the VTT cache noted earlier are included to guarantee the availability of at least one empty VTT cache line even in the worst case VTT occupancy scenario (since these lines can have no counterparts in the data cache). One of these lines is used by the cryptographic refresh process to hold the VTT line covering data lines currently being processed. Using this arrangement the refresh process accesses the VTT in the same way as the data cache. Even the VTT is refreshed in the usual way, resetting the root table entries as each line of the VTT is refreshed.

Thus a data cache miss that generates a VTT cache miss experiences an added delay that includes the time it takes to locate a free or freeable VTT cache entry plus a **Read** or a **Read** and a **Write**, for a clean or dirty VTT cache miss respectively. This added delay could easily increase the time required to satisfy a data cache miss by a factor of 3 or more. Hence differences in performance between a TRM-resident VTT design and a VTT cache design spring from two sources: the extra lookup associated with each data cache miss (to determine if the required VT is in the VTT cache and to ascertain its location if present) and the added delays resulting from VTT cache misses. The extra lookup step results in an increase of about 11-27% in effective memory access on a **Read**, versus 8-18% for a TRM-resident VTT, assuming primary memory chips are used for the VTT cache and tables or the resident VTT. Use of cache memory chips for the VTT cache and tables would equalize this difference between the two designs, based on a twofold access time improvement as a result of using the faster memory chips.

Since the VTT cache represents a relatively large percentage of the VTT for most systems (from 50% for a 256K-byte system to 12.5% for a 1M-byte system), its hit rate should be very high (on the order of 98% or more) and the added delays on VTT cache misses should constitute a negligible increase in effective memory access time. Thus the TRM-resident VTT offers design simplicity and good performance at the expense of a larger TRM, whereas the VTT cache engenders a complex design and reduced performance but a more compact TRM. Considering the complexity of the control logic for the VTT cache, it is not clear where above the 128K-byte primary memory size the breakeven point in TRM size lies between the two designs, especially if less dense high speed memory chips are used to improve performance of the VTT cache design. Thus the choice between a TRM-resident or encached VTT is not clear. The following descriptions of encrypted primary memory I/O assume the existence of a TRM-resident VTT to simplify the discussion. However, the differences that would result if the encached VTT design were adopted are noted and timing for the encached VTT design are provided in parentheses.

4.5.4 Encryption and EDC Calculation for Cache Lines

The cryptographic methods employed for T&A and secondary storage are not suitable for encrypted primary memory. In most computer systems the fetch of a cache line begins with the requested word (doubleword), which may not be the "first" word of the line, in order to minimize the delay associated with a cache miss. Any cryptographic method employing chaining imposes an ordering on the decryption of data and this is incompatible with the mode of cache operation cited above. Moreover, the minimum 5-cycle delay imposed by block mode decryption is at odds with this low-delay approach to satisfying cache misses. This suggests that the stream cryptographic method employed in the encrypted bus approach may be appropriate here. For encrypted primary memory, the cryptographic bit stream will be based on the IV formed from the cache line VT and the primary memory address, rather than on a counter and bit stream ID used in the encrypted bus approach. (Combined, the VT and address contribute about 36 bits to the 64-bit IV with the remaining 28 bits supplied by a fixed, per-TRM constant, just as in secondary and T&A storage.) This choice of IV limits pre-computation lead time since the bit stream cannot be calculated until the address and VT of the cache line are known, but the resulting delay is still better than that available through block modes.

This stream cryptographic method provides no propagation as an aid in detecting modification, so a separate EDC must be calculated. In the encrypted bus approach, a shortened (5 round) DES calculation was performed on the data and its address and the resulting CEDC was concealed for transmission under stream encryption. In the encrypted primary memory context, the doublewords that comprise a cache line are processed using the shortened DES calculation to yield four, 64-bit, preliminary CEDCs. These preliminary CEDCs must be combined to yield a 16-bit final CEDC that detects not only modification of individual doublewords but also

positional modification on doubleword boundaries, i.e., permutations of the doublewords in the line. This requirement is met by selecting 16 bits from each preliminary CEDC, concatenating them in an order based on positions of the doublewords in the cache line and processing this 64-bit quantity through a shortened DES. The final CEDC consists of 16 bits selected from this last processing step. This CEDC is concealed in the CEDC table in primary memory under stream encryption using the address of the CEDC and the cache line VT as an IV.

It is instructive to note why this particular method was chosen to calculate CEDCs for cache lines. The final CEDC could have been formed by chaining together the CEDC values from the cache lines, as was done in the *aggregate secure* transactions described in section 3.4.1. That method involves one (shortened) crypto operation per doubleword, four for the eight-word lines used here, and thus one might expect improved performance since the method proposed here requires five (shortened) crypto operations. However, on a **Read** of a cache line, the words in that line are fetched in an order determined by which word caused the miss. If the CEDC calculation was based on the chaining method used earlier, the calculation could not even begin until the first word of the cache line arrived. The CEDC calculation method adopted here is independent of the order of arrival of the words in the line and thus does not encounter delays of this sort. These considerations guided the choice of CEDC calculation methods.

The preceding descriptions of encryption and CEDC calculation are utilized in **Read** and **Write** operations in the following fashion. First consider a **Read** operation, i.e., the response to a cache miss or the first step in the refresh of a cache line, as depicted in Figure 4-6. The operation begins with transmission of the address for the doubleword containing the requested data (T1) and the lookup of the VT associated with the cache line containing that doubleword (A1). In an

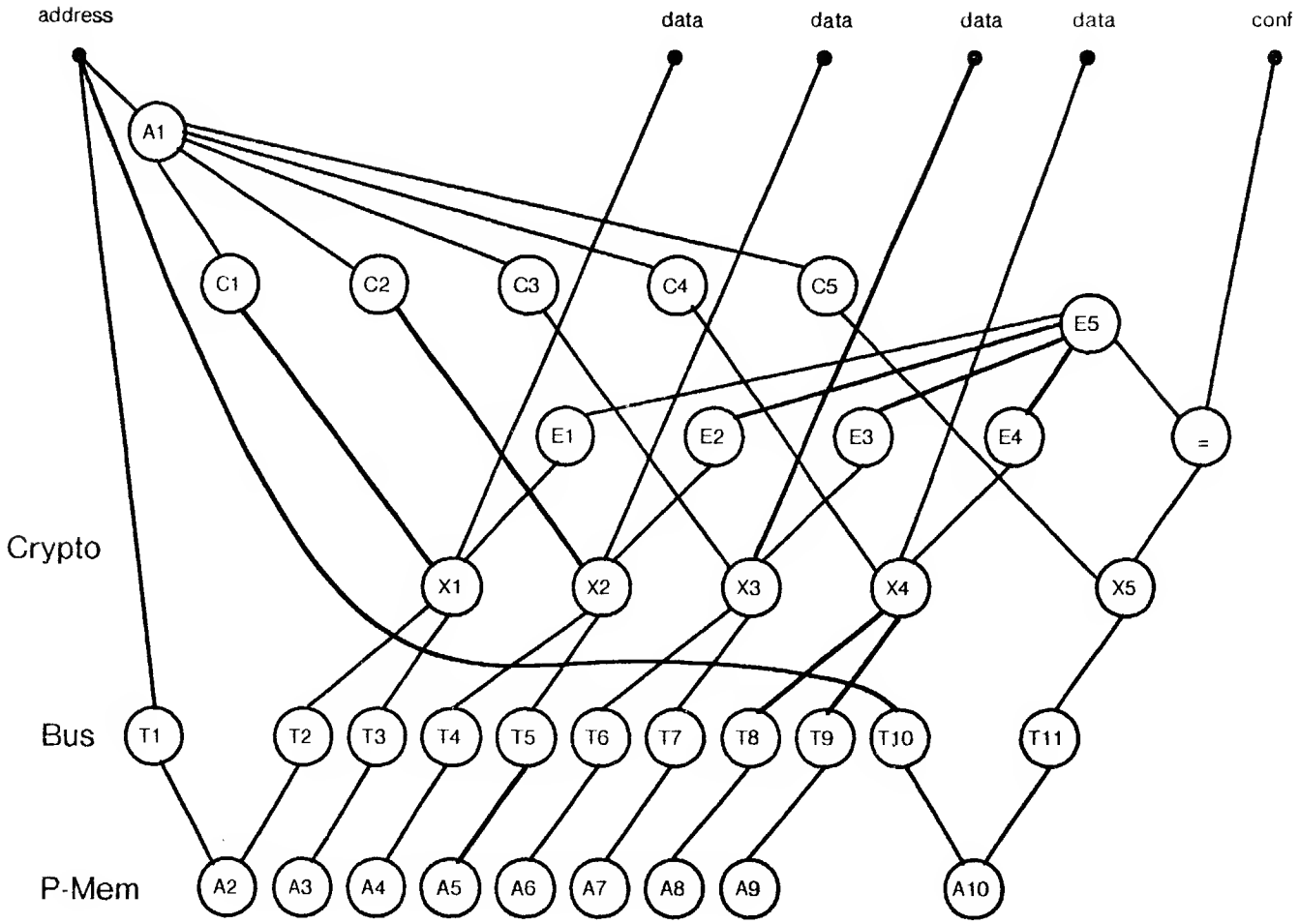


Figure 4-6: Event Graph for a **Read** of an Encrypted Cache Line

encached VTT design two lookups take place and, to minimize delay, the operation proceeds under the assumption that the required VT is in the cache. If a VTT cache miss occurs (detected after the first lookup), the request to primary memory for the data line is aborted and the VTT miss is processed. The fetching and transfer of the cache line begins with the doubleword containing the requested data and proceeds through increasing addresses, modulo the cache line length (A2-A9,T2-T9). Cryptographic bit streams for deciphering the cache line are generated using the

cache line VT and the addresses of the doublewords in the line (C1-C4). These bit streams are combined (via modulo 2 addition) with the doublewords transferred from primary memory to effect decryption (X1-X4).

Each decrypted doubleword is delivered to the cache and delivered for the preliminary CEDC calculations (E1-E4) and the result is processed to yield the final CEDC (E5) as described above. The stored CEDC is retrieved using a normal (not extended) bus transaction directed at the appropriate CEDC table location (T10,A10,T11). The bit stream for the CEDC is generated using the VT and the word address of the CEDC (C5) and is combined with the halfword containing the CEDC (X5). This decrypted quantity is compared against the calculated final CEDC to verify the authenticity, integrity and timeliness of the retrieved cache line.

Figure 4-7 presents the timing diagram for a Read of an encrypted cache line. Crypto devices 1-4 calculate the cryptographic bit stream and the preliminary CEDC for the cache doublewords and device 5 calculates the final CEDC and generates the bit stream to conceal this CEDC. The staggering of these processing steps may be used to reduce simultaneous demand on internal busses; it is esthetically appealing and is consistent with the precedence graph. In this diagram the fetch of the VT is accorded two cycles but, if a VTT cache is employed, the VT fetch time would increase to four cycles, even on a VTT cache hit.[12] The requested data is available 7 (9) cycles after the operation begins, the CEDC is available after 14 (16) cycles and the bus is busy for 13 cycles. The delay on data delivery is 4 (6) cycles greater than in a standard system or a comparable encrypted bus configuration and the CEDC delivery delay is 9 (11) cycles greater than in such an encrypted bus design. Bus utilization increases by 30% (3 cycles) over a comparably

[12]The parenthesized figures throughout the remainder of this section indicate the timing for systems with a VTT cache, assuming a hit on that cache.

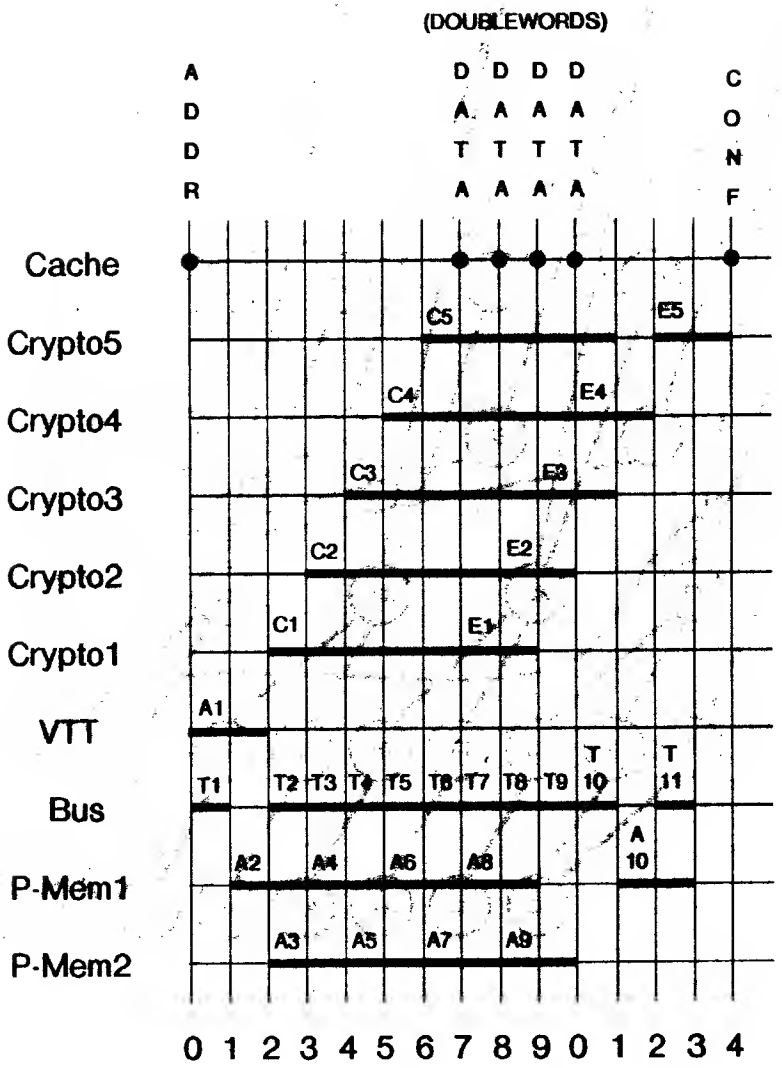


Figure 4-7: Timing Diagram for a **Read** of an Encrypted Cache Line

configured standard system but, since utilization is very low in these systems, this increase is significant only if it delays the initiation of another **Read**. Since the mean time between misses is expected to be on the order of 50-125 cycles (95-98% hit rate and average instruction length of 2.5 cycles), this delay probably has a negligible impact on system performance.

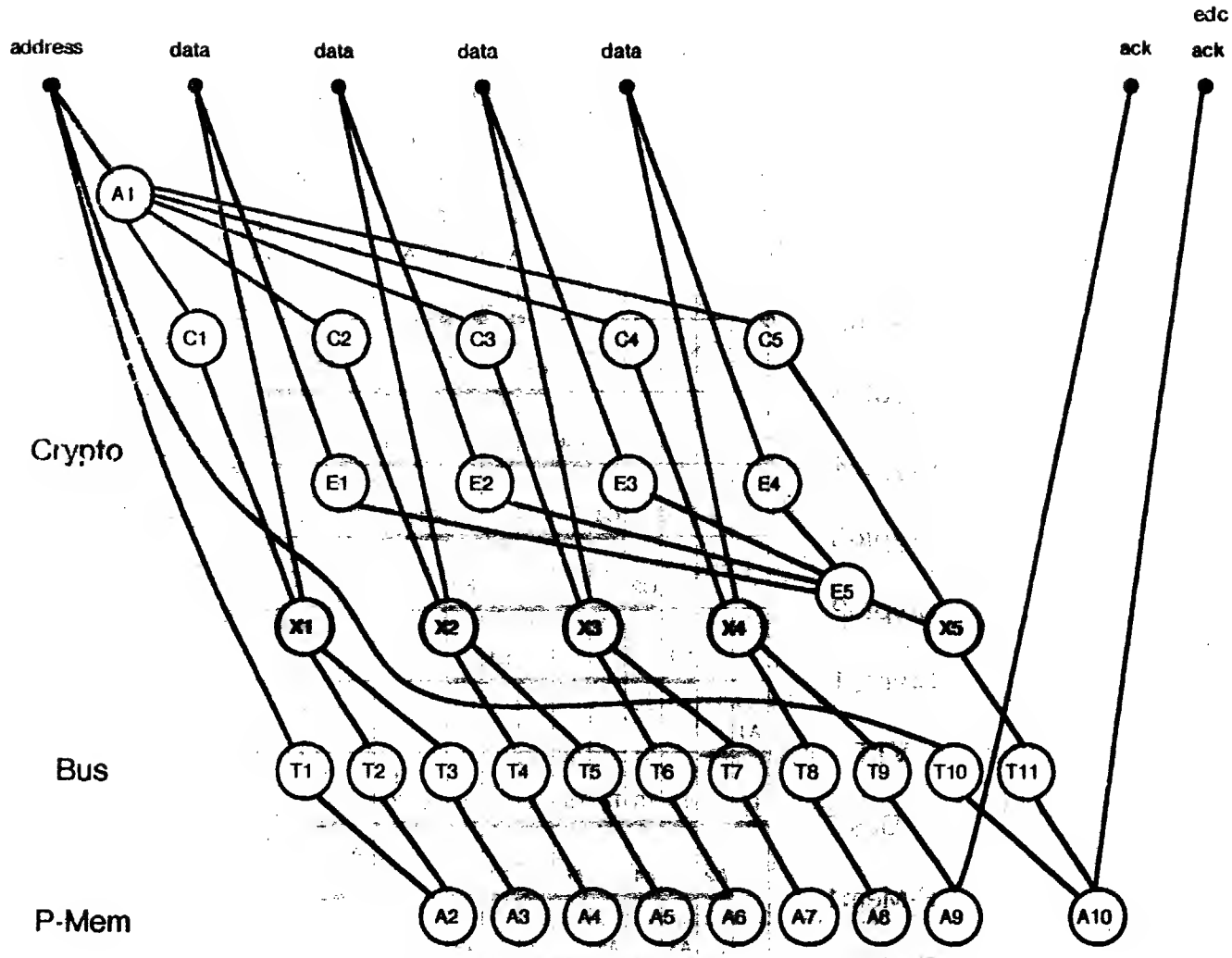


Figure 4-8: Event Graph for a Cache Line **Write**

Now consider a **Write** operation, i.e., the eviction of a modified cache line or part of a cache line refresh, as depicted in Figure 4-8. When a cache miss results in the eviction of a modified line the evicted line is buffered, the requested line is **Read** and then the **Write** of the evicted line takes place. This strategy results in all cache misses delivering the requested data after the same delay, even if a write-back is

required, unless buffer space for evicted lines is exhausted [6]. The operation begins with the lookup and update of the VT for the evicted cache line (A1). This VT is combined with the doubleword addresses of the line and used to generate cryptographic bit streams (C1-C5) for concealing the data and the CEDC. The doublewords (in increasing order) are combined with the bit streams (X1-X4), transmitted and stored in the appropriate memory locations (T1-T9, A2-A9) and acknowledged (T10). The preliminary CEDCs are calculated on these doublewords (E1-E4) and the results are used to calculate the final CEDC (E5) as described above. The final CEDC is concealed by combining it with 16-bits from *C5*, and the resulting halfword is transmitted and stored in the CEDC table (T11-T12,A10).[13]

Figure 4-9 presents the timing diagram for a **Write** of an encrypted cache line. The crypto unit utilization is the same as for **Read** operations. This operation requires 19 (21) cycles to complete and the bus is busy during the last 13 of those cycles. This operation is 9 (11) cycles longer than a cache line write in a standard system and 6 (8) longer than in a comparable encrypted bus system. Bus utilization is 30% greater than for a standard system and about 8% greater than for an encrypted bus system. (These figures assume the encrypted bus system incorporates separate bus lines for CEDC transmission, whereas the encrypted storage design employs a standard system bus.) As long as **Write** operations are adequately buffered, the added delay should not adversely affect performance. Again, given the very low bus utilization characteristics of these systems and the large mean time between misses, the additional bus cycles consumed for these operations should not significantly affect performance. Since most **Write** operations result from evictions triggered by **Read** operations, Figure 4-10 shows how the two operations mesh when

[13]There is a potential problem here in that only the halfword containing the CEDC for the affected cache line should be modified. If the primary memory does not support this form of partial word modification, then the whole word must be fetched, the relevant halfword modified and the whole word stored, increasing bus utilization and the effective cycle time for the **Write** operation.

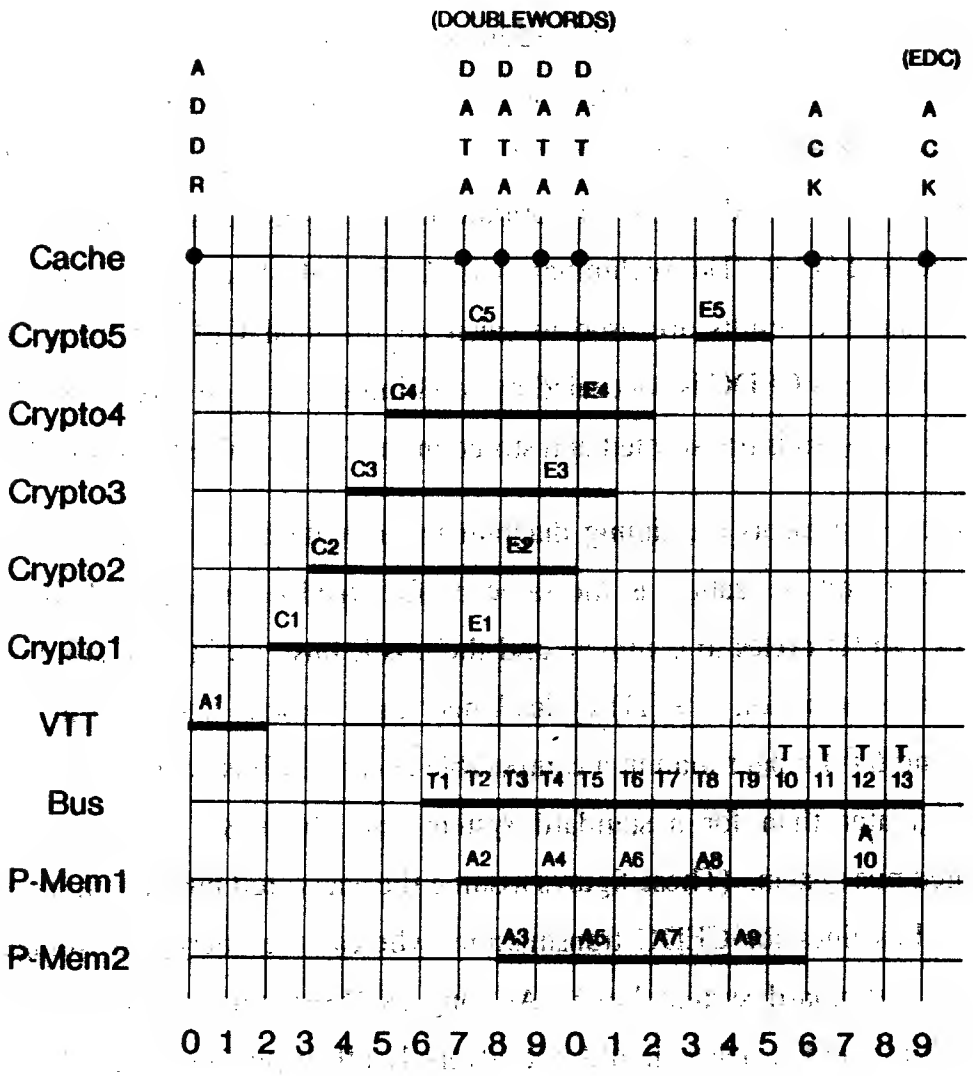


Figure 4-9: Timing Diagram for a Write of an Encrypted Cache Line

combined. Note that the total time for the combined operations is less than the sum of the independent operations due to overlap in processing steps.

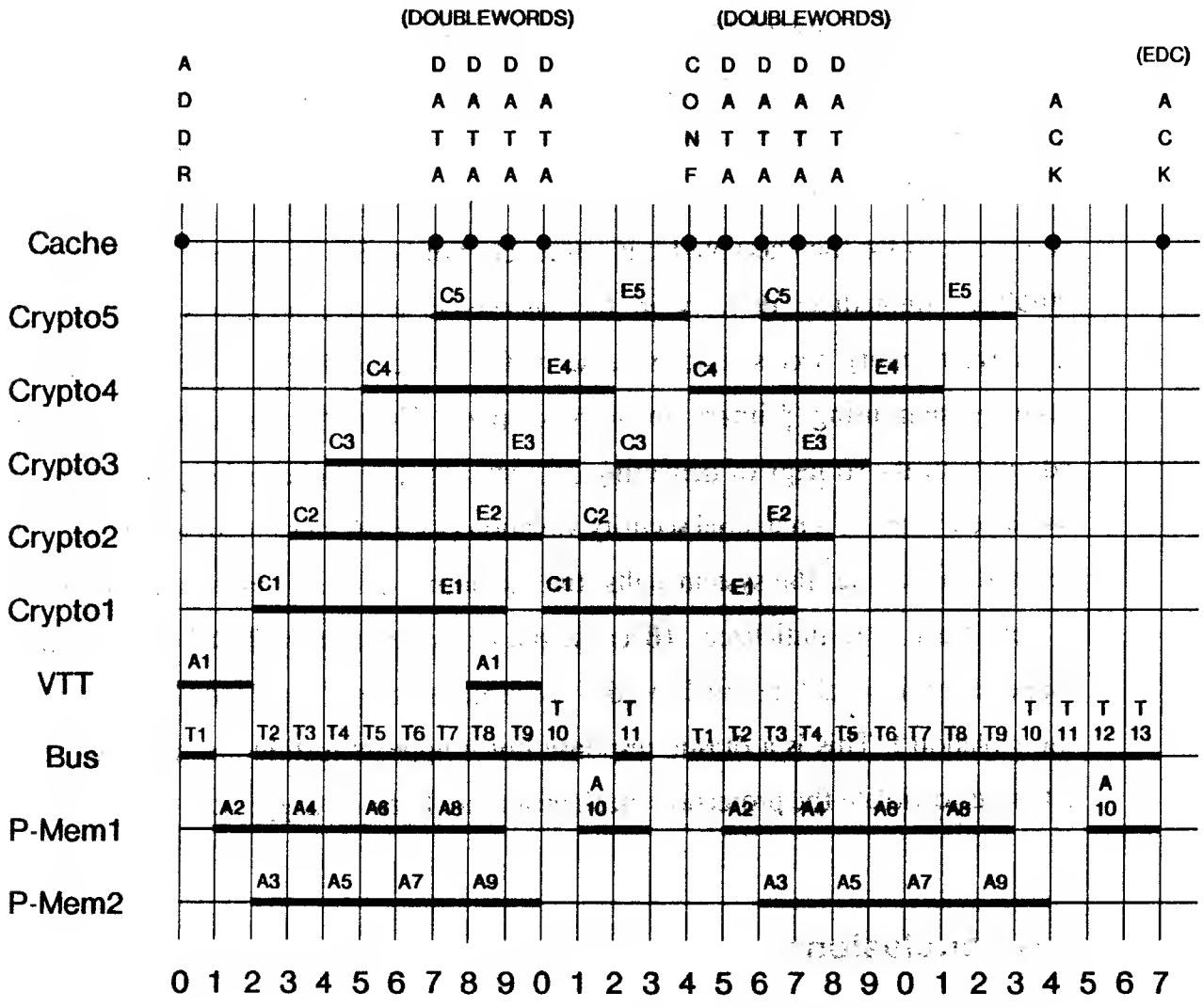


Figure 4-10: Timing Diagram for a Combined Read-Write Operation

As in the encrypted bus approach, there is a choice between delivering requested data immediately or deferring delivery until the CEDC is checked. However, in this case the CEDC is associated with the entire cache line, not individual words, and thus cannot be checked until the entire line, and the CEDC, have been transferred and decrypted. The increase in apparent memory access time associated with deferred delivery amounts to only 4-9% (for cache hit ratios of 98% and 95% respectively) for the encrypted bus approach, but here it would be anywhere from 20-50%. Immediate delivery in the encrypted storage approach results in an effective memory access time increase in the range of 8-18% (11-27% for a VTT cache design using primary memory chips). These figures strongly motivate adoption of the strategy of delivering data immediately and checking the CEDC on a delayed basis. If a potential security violation (a CEDC mismatch) is detected on a fetched cache line, the system halts, the violation counter is incremented and the system must be re-initialized. (Because the delay before the CEDC check is much longer here, it would be much harder for a processor to "back out" in response to the violation.) This is a drastic response but it appears justified as only deliberate attempts to violate the protection mechanisms are likely to trigger it.

4.6 Conclusions

The techniques developed in this chapter enable a computer system constructed using a single TRM and off-the-shelf storage devices outside that TRM to protect externally supplied software from disclosure and undetected modification. Several important concepts were introduced in this chapter to achieve this goal. Two concepts are fundamental to the protection mechanisms employed at all levels of storage. The first is the use of version tags (VTs) to form version-differentiated names for cryptographically transforming storage units. The second is the use of a protected version tag table to provide a basis for verifying the timeliness of storage

units on **Read** operations. For transfer and archival storage, the archival VTT and its associated update table provide a robust mechanism for enforcing reloading constraints for most-recent-only and non-reloadable files. The four-level hierarchic decomposition of the secondary storage VTT and appropriate caching of portions of this hierarchy makes the use of encrypted secondary storage feasible. Finally, cryptographic refresh for encrypted primary memory permits the use of small VTs with cache lines, significantly reducing the amount of memory devoted to security overhead.

The encrypted storage approach offers a number of advantages over the encrypted bus approach, especially in configurations such as **SYSTEM E** and **SYSTEM F**. Only with the adoption of encrypted storage techniques does secure T&A storage and demountable secondary storage become really practical. Off-the-shelf, demountable magnetic media are supported directly in this approach for these levels of storage. The only special requirement for these media arises in the secondary storage context where sector size must be increased slightly. However, most media are readily formatted to accommodate the larger sector size, so this is not a problem in most cases. The storage overhead for EDCs and VTs is small for both T&A and secondary storage, so this penalty should be quite acceptable. Management of the archival VTT is simple and should not perceptibly affect performance. The secondary storage VTT hierarchy requires more sophisticated management but still should not degrade system performance noticeably if primary memory is expanded to accommodate the VTT caches.

These security measures for T&A and secondary storage provide reduced cost and increased flexibility with only minor storage and performance overhead compared to comparable encrypted bus measures. The only significant potential drawback associated with these encrypted storage techniques is the loss of transparency, i.e., these techniques do require significant participation by the TRM

operating system. However, this disadvantage seems small compared to the advantages offered by this approach. At the encrypted primary memory level the storage overhead and performance degradation are more severe and the complexity of the TRM increases significantly. The cost of **SYSTEM H** in the encrypted primary memory approach may be comparable to that of **SYSTEM D** in the encrypted bus design due to this storage overhead and increased complexity, so the choice in this case is not so clear. Of course, **SYSTEM H** does offer greater flexibility in primary memory configuration and maintenance, but the comparison between the two configurations is complex. Perhaps a more important question is whether either **SYSTEM D** or **SYSTEM F** is preferable to **SYSTEM H.**

A major motivation for the adoption of **SYSTEM H** over **SYSTEM F** is the reduced size and cost, and presumably increased reliability, of the TRM in the former system. Of course there are other reasons for employing encrypted primary memory, e.g., increased flexibility in configuring and maintaining primary memory, but these are secondary in many applications. However, moving primary memory out of the TRM requires the addition of another SSI involving five crypto chips and control logic to support cryptographic refresh. It requires storage within the TRM either for the whole encrypted primary memory VTT or for the VTT cache and auxiliary tables and not inconsiderable control logic to manage the cache. Finally, this configuration requires the inclusion of a data cache and control logic which might not otherwise be required to achieve acceptable performance.

Since the crypto chips are very large compared to memory chips and the control logic chips also take up considerable space, the TRM space savings achieved by removing primary memory must be carefully analyzed. For many applications very large primary memories are not required and the ability to extend primary memory while retaining the same processor is not critical. For these applications a TRM configured with internal primary memory and encrypted secondary storage may be

preferable as the TRM would not be any larger and would probably be more reliable than a TRM for an encrypted primary memory configuration as described in section 4.5. The amount of primary memory that can be accommodated in the void left by the security hardware and data cache depends on the level of integration employed for the control logic and crypto chips and the density of primary memory chips. Using 256-Kbit primary memory chips and custom VLSI for the control logic and crypto chips, one could probably fit 256-512 Kbytes of primary memory in this void.

Finally, one can imagine hybrid designs employing a combination of the encrypted bus and encrypted storage approaches. Due to the difficuly of TRM-packaging of demountable media, T&A and secondary storage are probably better implemented using encrypted storage techniques. Yet, one might wish to conceal addresses on processor-generated references to primary memory (to minimize traffic analysis) and that is available only through the use of encrypted bus techniques. Thus, one might design a dual bus system in which primary memory is TRM-packaged and encrypted bus techniques are employed to protect traffic on the memory bus while encrypted storage techniques are used to protect data in secondary and T&A storage devices on the I/O bus. However, the cost of providing separate, TRM-packaged primary memory (as in **SYSTEM D**) is probably even greater than providing encrypted primary memory (as in **SYSTEM H**), since about twice as many crypto chips are required in the hybrid system. Thus, as in the preceding analysis, it is probably more feasible to incorporate primary memory into the main TRM (as in **SYSTEM F**) to achieve the required protection.

Chapter Five

Multi-Vendor Systems and Client Security Requirements

Chapters 3 and 4 developed several designs that meet the security requirements of the vendors of external software, i.e., encapsulation of external software to protect it from attacks resulting in the release or undetected modification of information. These designs assume that all external software executing on the TRM-packaged computer was supplied by a single vendor, i.e., the designs do not address the problem of multi-vendor computer systems. Moreover, these designs do not address the security requirements of the clients of external software, i.e., confinement of external software to prevent disclosure of client-supplied information to the "outside world" and to control access of external software to computer resources not devoted exclusively to the vendor of that software. These two problems can be unified by viewing the client as a vendor possessing certain extra privileges, e.g., control over access to shared system resources. This chapter explores the problem of designing systems that support client security requirements and external software supplied by multiple vendors. It examines two approaches to solving this problem: use of third-party supplied TRMs equipped with secure operating systems and multi-TRM systems.

5.1 Confining External Software

Since the computer systems of interest are under the direct physical control of the clients, leakage of client-supplied information outside of the client-controlled environment takes place only through communication with the outside world. The primary channel for such leakage is the communication network interface. Other channels may exist as well, e.g., hardcopy output circulated outside the client environment and maintenance by external vendor personnel, but these are dealt with by procedural rather than technical security controls. Some personal and small business computers will not have a network interface, effectively eliminating this leakage problem. However, distributed systems and many personal and small business computers will have network interfaces and the problem of leakage will arise.

The level of difficulty associated with preventing leakage of client-supplied information depends on the configuration of the computer system and what use external software makes of network communication facilities. In order to restrict access by external software to a network, the client must have direct control over the network interface. If a client's only means of controlling this interface is through a processor and/or software provided by an *untrusted* vendor, e.g., the vendor supplying software that is to be confined, then confinement cannot be achieved. However, a client exercising direct control over this interface can prevent or at least minimize leakage of his data in many circumstances. If external software does not use the network as part of its normal operation, then client-controlled security mechanisms can prevent the software from accessing the network at all. If external software uses the network only in a very restricted fashion, then security controls can mediate access to the network to prevent or severely restrict leakage.

5.1.1 Preventing Information Leakage in Simple Applications

Consider, for example, external software that establishes a connection to a service that provides current stock quotations or other information based on a tightly constrained query set. This type of external software can be confined reasonably well since the flow of information is essentially one-way (from the service to the external software). Despite the one-way nature of this sort of communication, external software might try to leak information by signalling over *covert channels*, e.g., manipulation of connection flow control parameters, since network protocols do involve some *reverse* flow of information even for one-way data transmission. The rate at which information can be leaked in this fashion can be made arbitrarily low if the communication protocol is not implemented by external software but rather is under client control. A connection-oriented data transport protocol (see section 2.3.4) supplied and controlled by the client would be an appropriate interface for much external software and would provide the client with control over many covert channels (for suitably constrained network usage).

Even the task of external software re-authorization, i.e., notifying the software that the client has paid the "rent" and thus the software should continue to operate, can be tightly constrained so as to minimize leakage potential (thus achieving a high degree of confinement). Simple re-authorization procedures do not require any transmission of data from the external software to the vendor. The software can maintain a counter of the number of times it is invoked and another counter that tracks *re-authorization notices*. Depending on the duration of the rental period and the nature of the subsystem, a limit is established as the maximum number of invocations allowed before re-authorization.[14] The vendor, upon receipt of periodic

[14] If a clock with battery backup could be included in the main TRM, reauthorization could be based on time (e.g., months) rather than on the number of times external software was invoked by the client.

payment, issues a re-authorization notice (incorporating an encrypted form of the re-authorization counter) to the client, who forwards it to the external software. The subsystem verifies the re-authorization notice, resets the invocation counter and increments the re-authorization counter. More elaborate re-authorization procedures might involve transmission of usage statistics by the external software, e.g., for billing purposes. The integrity, authenticity and timeliness of these statistics can be ensured by covering them and the re-authorization counter with a CEDC. This procedure minimizes leakage potential and thus should prove acceptable to clients.

5.1.2 Preventing Leakage in Distributed Applications

Security measures of this sort are sufficient for many of the proprietary software applications that use network facilities. However, in the context of distributed systems, one may encounter external software that engages in substantial, complex two-way communication among copies of itself implementing distributed applications at the nodes in the system. Automated mediation of this sort of communication to prevent leakage of client data is not feasible, both because of the complexity of the message exchanges and because the transmitted data may be encrypted by the external software copies to meet the security requirements of subsystem vendors. In the simplest case, clients may wish to confine external software to preclude leakage of information outside of the distributed system user community. This is readily accomplished since clients can superimpose their own inter-node communication security measures (using keys available only to members of the user community), on top of any communication security measures employed by external software.

However, as indicated above, if clients require a more sophisticated sort of confinement of external software, problems may arise. Consider, for example,

external software managing a distributed (but not replicated) database containing information supplied by various members of the distributed system user community. Each client may place constraints on how information supplied by him is made available to other clients, e.g., data private to each client may be maintained at his node and database access controls will allow him to restrict access to this data. Either the client can rely on the external software to enforce these controls or he can attempt to mediate inter-node communication involving the database management subsystem. In this situation automatic mediation is difficult at best and is impossible if external software uses encryption to conceal inter-node communication. Even if inter-node communication is not cryptographically concealed by the external software, e.g., the software employs cryptographic methods only for authenticity and integrity checks, strict mediation of inter-node communication would require duplicating the operation of the database subsystem. Yet such duplication by the client is in direct conflict with the acquisition of external software!

This problem worsens if clients must rely on a distributed subsystem to enforce access control policies for data dispersed throughout the system, e.g., fully replicated distributed databases containing sensitive client data. In this case, communication among copies of the subsystem may be encrypted by the subsystem (to conceal the client data transmitted between the copies), thus denying the client any opportunity of monitoring to prevent or even detect leakage! Clients might be able to trust external software to enforce an advertised access control policy if they, or a trusted third party, could inspect the source code and establish its correspondence to the executable subsystem installed at each node. Client inspection of proprietary software is not likely to be acceptable to vendors, but in the distributed system context, such inspection may be viable when external software is supplied by members of the user community. In the latter case, disclosure of the software within the user community is not a major concern but protection of the data managed by

the software must be ensured. What is required, however, is some means of establishing correspondence between the inspected and installed subsystem copies without compromising subsystem integrity and while providing for secure communication among subsystem copies. These requirements can be met using procedures described in the next section.

5.1.3 Controlling Access to Shared Resources

The other aspect of confinement is controlling access of external software to computer system resources not exclusively devoted to the vendor of that software. This security requirement is applicable only in computer systems which support secure execution of software from multiple independent vendors, possibly including the client himself. (In a single-vendor system all facilities are available exclusively for the use of software provided by that vendor and any sort of confinement beyond disconnection of the system from the network is meaningless.) Resources to which access may be controlled include portions of the storage hierarchy, the terminal and other I/O devices, e.g., the network interface. The guideline here is the *principle of least privilege* employed in secure system design, i.e., a subsystem should have access only to those resources required to carry out its designated function [29].

Access restriction of external software is important for several reasons. For example, access controls applied to external software often simplify the information leakage aspect of confinement since software can disclose only that information to which it has access. External software that has no access to sensitive client information poses no leakage threat and thus does not require the sort of network access mediation accorded external software that does have access to such information. If the latter software does not use the network and the former does, the leakage problem is significantly simplified. When secondary storage is shared, for example, software of one vendor must be prevented from damaging software of

another vendor (or of the client) and the quantity of storage consumed by external software should be controlled. With respect to the terminal, the client must be able to select and identify the software with which he is communicating in order to prevent confusion that could result in violations of client access controls. Finally, control of access to the network interface, as noted above, is the fundamental means by which the information leakage problem is managed. Thus, controlling access of external software to shared system resources really encompasses all aspects of confinement.

5.2 Computer Systems Supplied by a Third-Party

One way to accommodate software supplied by multiple vendors in a single computer system is to use one of the designs presented in Chapter 3 or 4 in conjunction with a secure operating system, with all security relevant hardware and software supplied by a trusted third party. The secure operating system performs two functions: it protects external software from attacks by other software (the security mechanisms of Chapters 3 and 4 protect against physical attacks) and it confines software to control information leakage. In single-vendor systems, the level of security required of the operating system depends to a great extent on the nature of the application software provided by the vendor. For example, external software implementing financial applications or games require less sophisticated protection mechanisms than external software controlling execution of client-written code on the vendor-supplied processor. In multi-vendor systems, the operating system must withstand programmed attacks mounted by vendor or client software in order to provide encapsulation and confinement of external software. Thus the level of operating system security required in multi-vendor computers is relatively high.

5.2.1 Options for Software-Enforced Encapsulation

In the extreme case, the operating system for a multi-vendor computer might provide a fine-grained *protection domain* structure that supports mutually suspicious subsystems while providing an invocation mechanism essentially equivalent to normal procedure calls (see [29]). Although several operating systems and machine architectures that implement this form of sophisticated protection have been described in the literature, few have been constructed and none are commercially available at this time. This type of operating system and its associated hardware support facilities are generally quite complex, in contrast to the simplicity that tends to characterize the computer systems of interest. Although it is conceivable that such sophisticated hardware and software could be provided in small, multi-vendor systems, it may not be necessary. For many applications, it is not critical that invocation of external software be as flexible and as fast as normal procedure invocation. For example, compilers, editors, games or financial application packages are not invoked with very high frequency; they execute for some time before completion and are unlikely to make extensive use of other subsystems. Thus a facility that supports mutually suspicious subsystems but provides a somewhat less convenient interface than normal procedure invocation might be appropriate in many circumstances.

A secure *virtual machine monitor* (VMM) [13] is much simpler to construct than a fully general protection domain system, yet it can provide the necessary encapsulation and confinement, albeit with less convenient invocation of external software. A multi-vendor system can be implemented by using a VMM in which each vendor is represented by a separate virtual machine implementing a very simple environment for external software development and operation. The VMM maps the system resources used by the virtual machines into physical resources. For example, the VMM partitions physical memory among virtual machines and may

map a selected portion of virtual machine memory to provide data transmission between the virtual machine and the VMM. Secondary storage may be provided by partitioning physical disks into *mini-disks* that are private to virtual machines (as in VM/370). The VMM intercepts I/O instructions and translates them so that accesses to a mini-disk are directed to the appropriate region of a real disk. Invocation of external software can be effected through inter-virtual machine communication. The VMM can provide communication among virtual machines in a variety of ways, e.g., by simulating network connections between the virtual machines.

To a great extent, encapsulation and confinement of external software are achieved by the implicit isolation of virtual machines provided by the VMM. The client, interacting with the VMM directly via his terminal, can act as a sort of limited system administrator as well as the *owner* of a virtual machine. This provides him with the tools necessary to control access to shared system resources, e.g., storage and I/O devices, but he is not granted the ability to examine unencrypted data internal to vendor virtual machines. The VMM design makes it especially easy for the client to control secondary and T&A storage usage and access to peripherals, since all physical devices are available to the virtual machines only through the explicit mediation of the VMM. For example, the VMM may interpret and translate control transactions involving DMA devices and other peripherals as a matter of course, and access control checking is readily incorporated into these activities. This design even allows the client to supply software for automatic mediation of network access in a fashion that is transparent to the vendor virtual machines, since the VMM mediates such access anyway.

The third-party design requires clients and vendors to trust the supplier of security relevant hardware (TRMs) and software to provide a product that meets the security requirements of both parties. It is likely that both parties will want to

inspect the software to satisfy themselves that it properly implements the encapsulation and confinement security policies described above. The simplicity and relatively small size of a VMM makes it more amenable to visual inspection and automatic verification, and that makes its acceptance by clients and vendors more likely. (The assumption here is that the third party will accept disclosure of the VMM design and code as a necessary part of his business.) Similarly, the hardware design and the TRMs must be available for examination. Assuming that these criteria can be met to the satisfaction of both parties, the major remaining question is how to distribute external software to these computers in a fashion that meets the security requirements of both clients and vendors.

5.2.2 Distributing External Software in the Third-Party Design

The simplest solution to the problem of distributing external software is to make the third-party supplier the distributor as well. Vendors could provide the third-party supplier with their software and he could securely distribute it to clients, possibly acting as a collection agent for the vendors as well. The distribution could be carried out using any of the methods described previously using conventional ciphers, e.g., encrypted transfer storage or secure down-line loading. This requires a high level of trust on the part of the vendors since their software is directly available to the third party, and the clients may be wary of this close relationship between vendors and the presumably impartial third party. Instead, an approach based on the use of public-key ciphers (PKCs) for external software distribution may prove more acceptable to clients and vendors. Using public-key ciphers, it is possible to eliminate the TRM supplier from the distribution procedure, so that only the vendor and the TRM-based computer have access to external software.

The public-key cipher distribution procedure operates in the following fashion. The third-party supplier provides a public-key cipher facility in a secure portion of each TRM-packaged computer system. This facility implements public-key cipher transformations and generates a PKC key pair for use in the secure software distribution procedure. After the computer is purchased, this key pair generation is carried out in the presence of the client and some independent agent that serves as a *registrar* of public keys for these third-party computers. (The third-party supplier might serve this function and additional witnesses may be present.) The client and the registrar both supply random inputs to the TRM for key generation, providing unbiased key selection, then they initiate the process. When the key pair is generated, the secret key is held in (erasable) non-volatile storage, never to be known outside the TRM, and the public key is output by the TRM. This public-key is recorded by the registrar, establishing the correspondence between it, the TRM-based computer and the client.

To distribute external software to this computer, a vendor checks with the registrar to establish the association between the public key and the computer in question. Using this public key, the vendor encrypts a (secret) conventional cipher key and an identifier, generated by the vendor, for use in secure down-line loading or for encrypted storage distribution. Once this initial contact has occurred, a vendor can identify himself to the third-party supplied computer in subsequent distribution procedures by using the same secret conventional key and identifier. The client interacts with the computer to establish his own subsystems in a more direct fashion based on his direct physical control of the system, e.g., through console interaction. Since the secret key of the PKC pair is known only to the TRM-based VMM, only the vendor and the TRM have access to software distributed in this fashion. Of course, this procedure is meaningful only if TRM-packaged system components are permanently sealed at the factory, i.e., not subject to subsequent invasive maintenance procedures. This strongly suggests the use of

an encrypted-storage design, e.g., **SYSTEM G** or **SYSTEM H** from Chapter 4, to minimize the number of TRM-packaged components.

This software distribution procedure based on public-key ciphers meets the needs of vendors of proprietary software for many applications. In distributed systems employing this procedure, members of the user community can act as vendors to exchange software in a fashion that protects the lender. However, this procedure does not address the special problem of distributed software that must be trusted to implement access control policies, e.g., the distributed, replicated database subsystem described above. If such subsystems are provided as proprietary software by a vendor, it is unlikely that inspection of the subsystem source code by the clients will be acceptable, so at best an independent party might be brought in to certify the *correctness* of such subsystems. If this certification procedure is acceptable to both clients and vendors, the subsystems can be distributed using the procedure described above. A vendor would associate a secret key with the subsystem copies destined for a given distributed system, providing them with a basis for secure inter-node communication. (The subsystem copies are identified to one another by the hardware UID associated with each computer.) If mode nodes are added to the distributed system, the vendor can supply additional copies of the subsystem with the same key.

5.2.3 Distributing User-Written External Software in Distributed Systems

If the subsystem is supplied by a member of the distributed system user community, the problem is somewhat different. The assumptions here are that the members of the user community will co-operate in this process and there is no requirement to conceal the subsystem code, but the users are largely autonomous and thus harbor some degree of mutual suspicion. Thus perspective clients

(members of the user community) may inspect the code to verify that it implements an advertised security policy. However, the user/vendor who wrote the subsystem cannot directly distribute the subsystem since he cannot be allowed to know a secret key embedded in the subsystem copies for secure inter-node communication. This problem can be solved by using a third-party computer with appropriate software as an *installation server* for the distributed system. This computer is a shared resource of the distributed system user community and is operated by them co-operatively. The installation server acts as a surrogate for user-vendors in carrying out the subsystem distribution process in a fashion that meets the security requirements of the user community. Readers not interested in the details of how this process is implemented should skip to section 5.3 (page 226) for a discussion of the other approach to realizing multi-vendor computer systems.

Figure 5-1 illustrates the flow of messages in this procedure, using an example distributed system composed of 4 user nodes (*A-D*) and an installation server node (*E*). The subsystem creator, in this example, user node *D*, initiates the procedure by transmitting a copy of the subsystem source code to the installation server node (*step 1*). This transmission is secured using the secret key of the third-party computer along with an EDC or AICF to ensure authenticity and integrity. The installation server records this subsystem, assigning it a UID, and compiles the subsystem, producing the executable object module version. Included in the object module is a secret key, generated by the server, which the subsystem copies can use to communicate securely with one another. The server distributes a copy of the source and object module versions of the subsystem to each user node (*step 2*); the source code is provided for the inspection and approval of the user and the object module is made available for immediate installation and activation of the subsystem. (Distribution of the subsystem can be restricted to a subset of the user community by informing the installation server of this subset at the time the subsystem is delivered by its writer.)

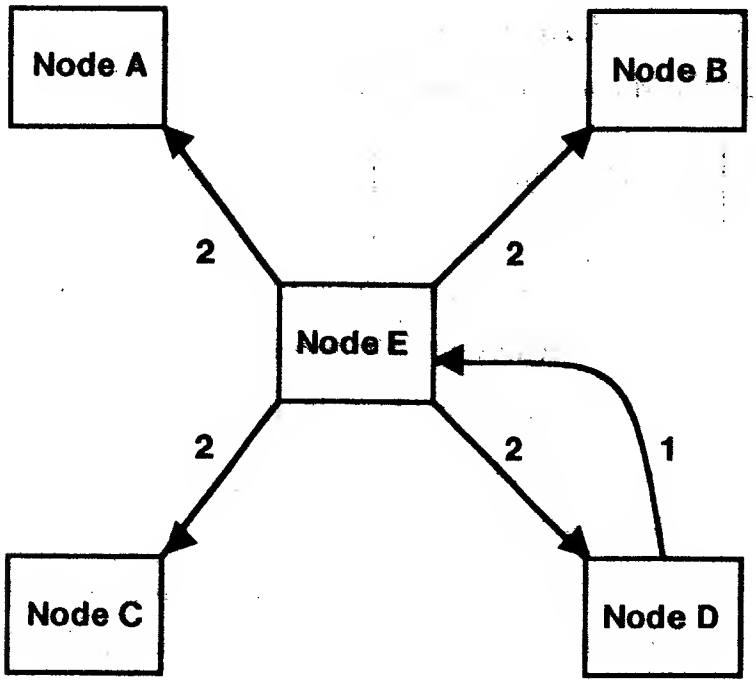


Figure 5-1: Secure Installation of a User-Written, Distributed Subsystem

Each transmitted copy of the source and object modules is transformed under the secret key of the installation server to ensure authenticity, then under the public key of the target user node for secrecy, and an EDC is included for integrity checking. In order to effect these transformations, the installation server must be provided (in a reliable fashion) with the public keys of all the user nodes. The public key of the installation server must be made available to the user nodes to allow verification of this transmission. (If a user node is provided with a public key that does not correspond to the installation server, the security of the procedure is not violated, but the node in question will not be able to decipher and load subsystems!) Each user node VMM, upon receipt, transformation and verification of this transmission,

makes available the subsystem source code for user inspection. If, after examining the source code, a user approves it, he authorizes his node VMM to install (and thus activate) the subsystem. Users not wishing to participate in the subsystem merely instruct the node VMM *not* to install the subsystem.

This procedure guarantees that the installed subsystem copies are identical, that they have been approved by the users (clients) on whose computers the copies are executing, that they can communicate securely with one another and that the subsystem writer cannot circumvent this procedure, i.e., he is bound by the advertised access control policy embedded in the subsystem! This is a simple procedure and, although it requires the users to exercise some care in operation of the installation subsystem, the procedure meets the stringent security requirements established for distributed systems composed of autonomously managed nodes.[15] Moreover, the installation procedure can be effected incrementally, i.e., members of the distributed system can participate in the installation and use of subsystems at their convenience. The introduction of a new node into the distributed system requires registering the node with the installation server, i.e., establishing the correspondence between the node UID and its public key, before subsystem copies can be installed at the new node. (This simple task requires supervision by the users to ensure that the proper public key is installed.)

5.3 Multi-TRM Computer Systems

Although the third-party computer approach meets the security requirements established for multi-vendor systems, it does require the vendors and clients to trust the third-party supplier. Moreover, it may require the supplier to disclose his

[15] Trojan Horse programs could still be a problem here, but at least the user can examine the source code (perhaps using program verification tools) in an attempt to locate any Trojan Horses.

hardware and software designs and make his system available for inspection in order to satisfy the concerns of the vendors and clients. The problems related to trusting a third-party supplier can be avoided if each vendor supplies his own security relevant hardware and software. This vendor-supplied hardware and software can be organized into a computer system that operates much like a distributed system in microcosm. Each vendor is represented by his own TRM (acting as a node) and the client controls interactions among these nodes and access to shared system resources. In this fashion each vendor is responsible for meeting his own security requirements through the hardware and software encapsulation mechanisms he provides, and the client confines the external software through the use of hardware and software that is completely controlled by him. This approach retains the simplicity of single-vendor systems yet provides the functionality of multi-vendor systems as achieved in the third-party VMM design.

5.3.1 Configuration Options for the Multi-TRM approach

The primary drawback associated with this approach is the cost of providing duplicate TRM-packaged hardware, one system per vendor. However, if the cost of these systems can be made sufficiently small relative to the anticipated revenues from sales or rental of proprietary software, this approach may be economically feasible and acceptable to both vendors and clients. The need to minimize costs strongly suggests the use of encrypted storage designs since they involve only one TRM and can share storage outside the TRM. The TRM designs of **SYSTEM G** and **SYSTEM H** are the most promising candidates as they yield the smallest, least expensive TRMs and offer the greatest opportunity for storage sharing. Using either design, the (vendor-supplied) TRMs share secondary and T&A storage and I/O devices (terminal, net interface, etc.) under client control. Using the design of **SYSTEM G**, primary memory is shared only as a medium for parameter

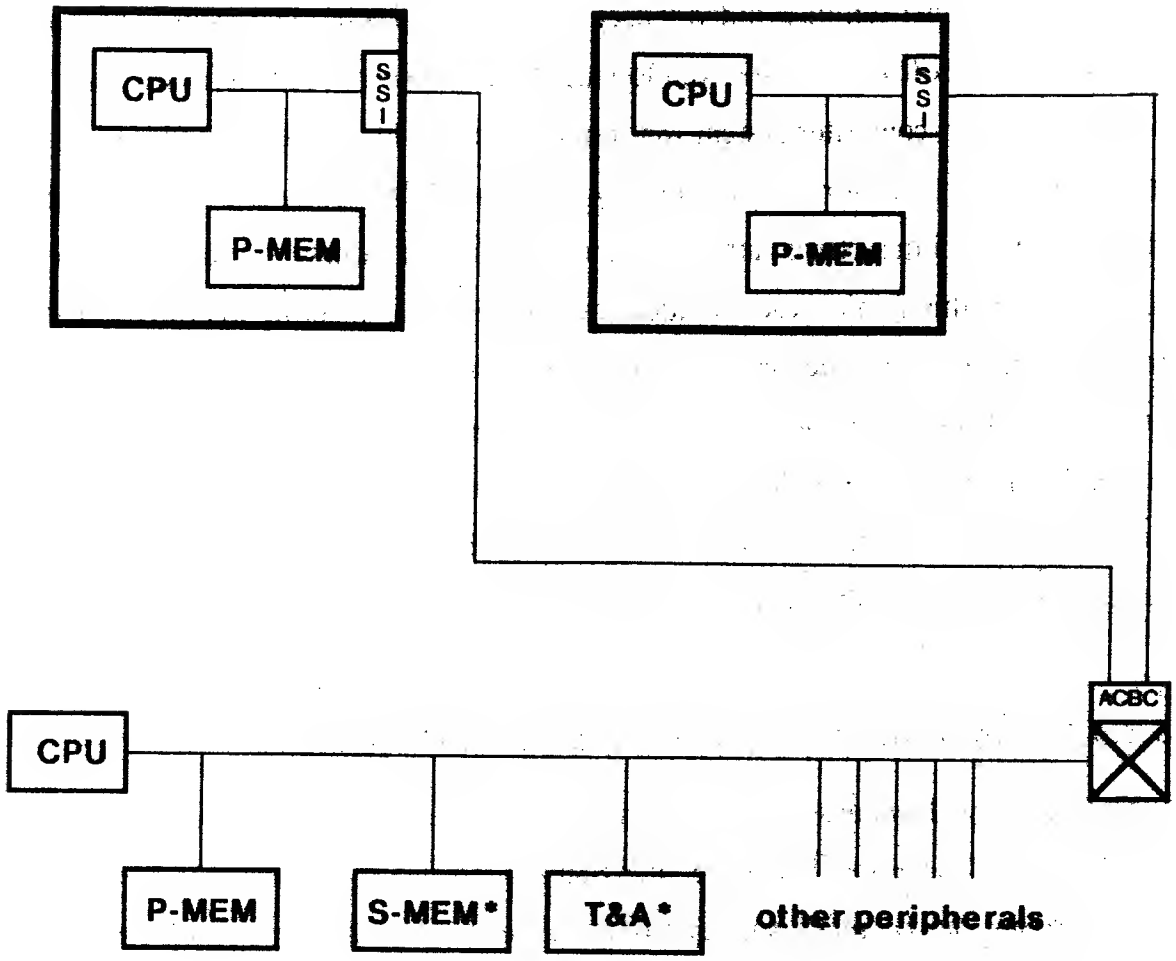
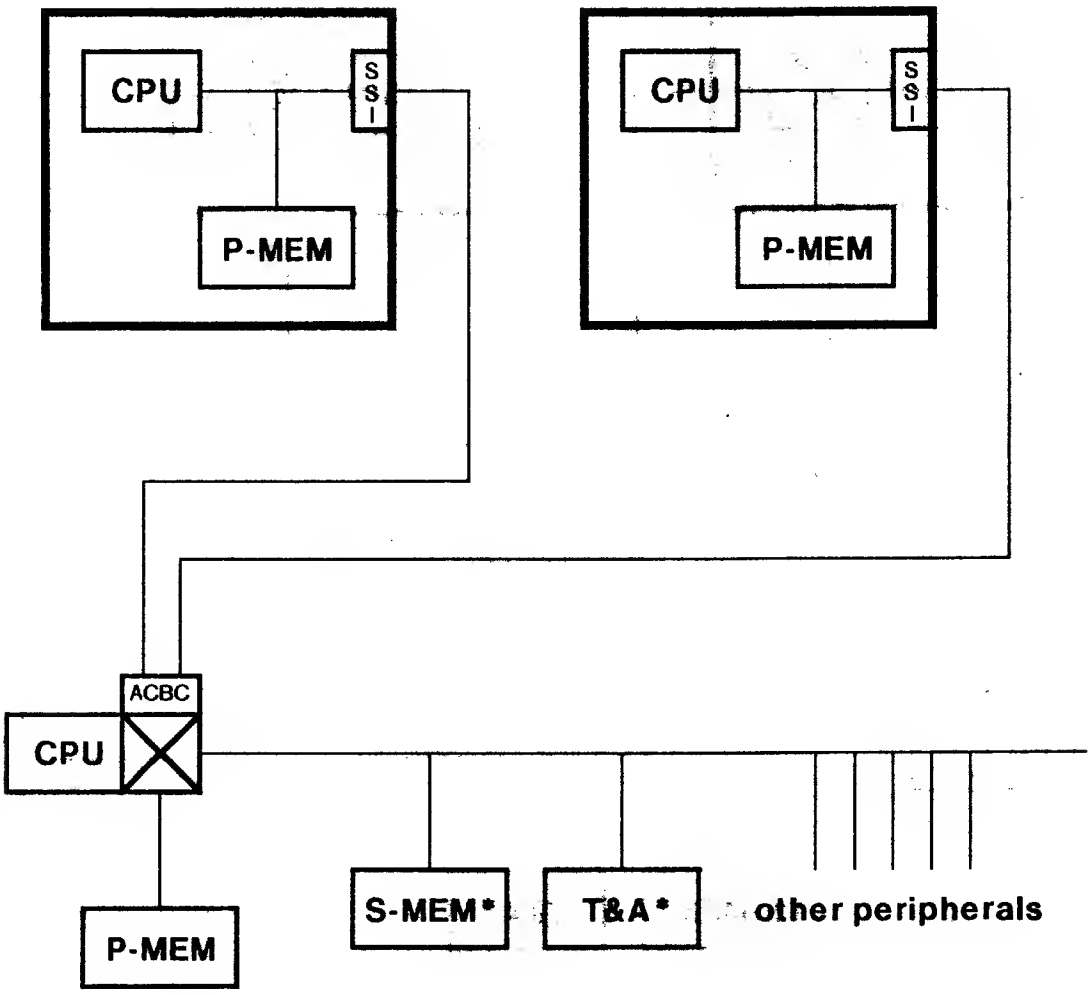


Figure 5-2: A Single Bus Multi-TRM System Configuration

transmission between processors, i.e., physically unprotected primary memory is provided primarily for use by the client-supplied processor since each TRM contains built-in primary memory. A multi-TRM system based on the design of **SYSTEM H** could share all primary memory among all the processors (client and vendor). Figures 5-2, 5-3 and 5-4 show three multi-TRM system configurations.



System J

Figure 5-3: A Dual Bus Multi-TRM System Configuration

The first two configurations, **SYSTEM I** and **SYSTEM J**, illustrate TRMs with built-in primary memory connected to single and dual bus systems, whereas the third configuration, **SYSTEM K**, shows TRMs sharing primary memory with the client processor in a dual bus system. All three configurations require essentially the

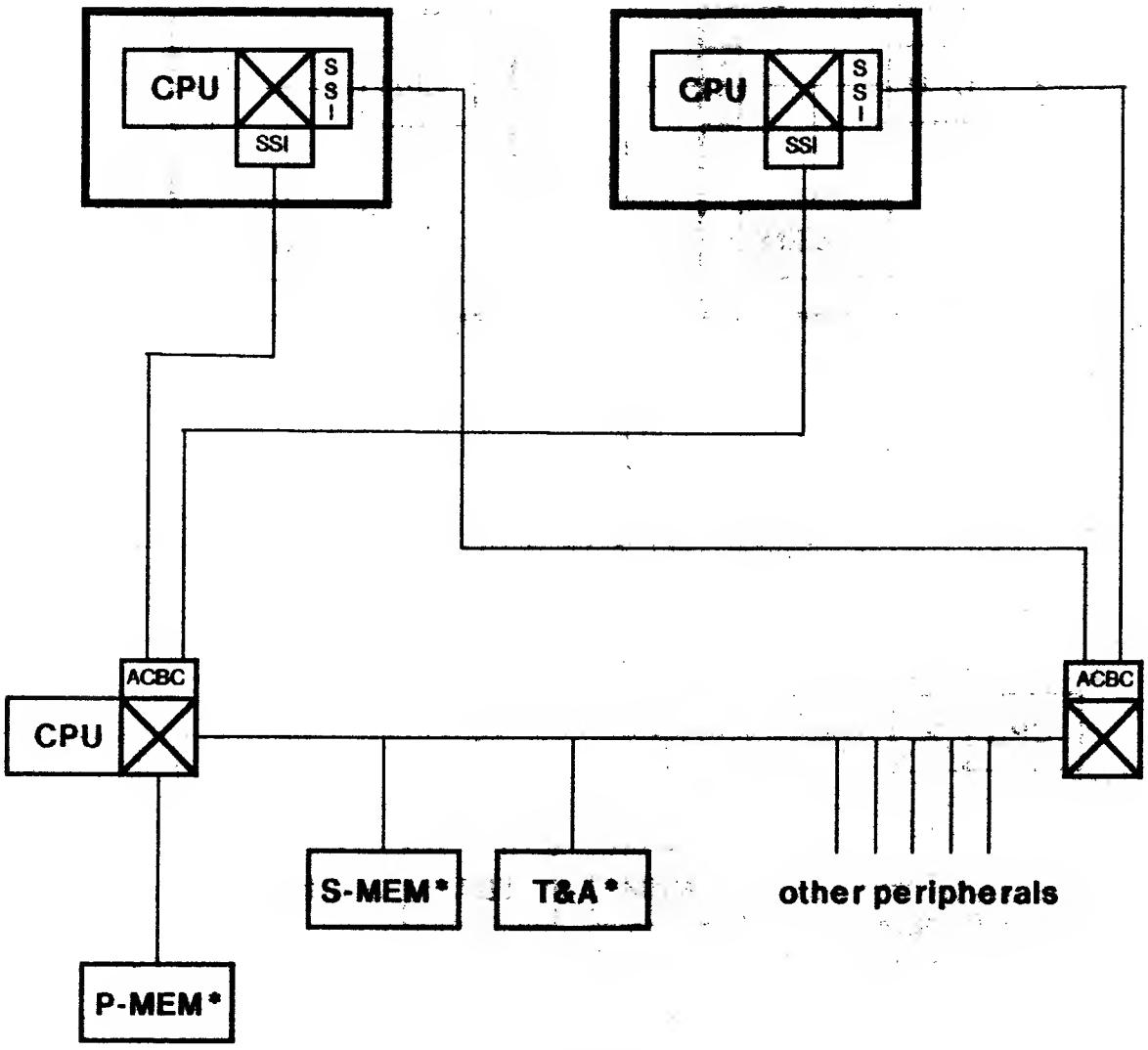


Figure 5-4: Another Dual Bus Multi-TRM System Configuration

same access control mechanisms to enforce confinement of external software. (Remember, encapsulation is provided by the TRM-packaging and encrypted storage security mechanisms described in Chapter 4, both of which are vendor-

supplied). The access control requirements here are generally the same as in the VMM design and the mechanisms used to achieve them may be quite similar; only the implementation of the mechanisms is different here. In order to maximize the use of off-the-shelf system components, e.g., disks and I/O devices, an *access control bus coupler* (ACBC) is employed to connect TRM bus(ses) to the main system bus(ses). The alternatives, enforcing access control at the bus interface to each shared resource or at each TRM-bus interface, would require additional specialized hardware. Moreover, access control hardware may introduce some delay in bus transactions and the ACBC design imposes this delay only on accesses to shared resources by TRMs, i.e., it need not affect performance of the client processor.

An ACBC is the dual of a secure bus coupler (SBC), i.e., the ACBC protects client equipment from attacks by vendor TRMs in much the same fashion that an SBC protects TRM-packaged vendor equipment from client attacks. An ACBC filters traffic on the bus(ses) connecting shared resources and the client-supplied processor, so transactions local to those components are not repeated on the TRM bus(ses). The ACBC also controls TRM access to primary memory, secondary and T&A storage devices and various I/O devices, e.g., the terminal and the network interface, as directed by the client. To properly enforce access control, each TRM must be reliably identified to the ACBC and confinement requires that transactions involving one TRM must not be passively or actively wiretapped by other TRMs. One cannot simply connect multiple TRMs to a single, conventional bus since such a bus does not preclude passive and active wiretapping attacks by other TRMs on that bus. Thus each TRM has its own short bus segment(s) connecting it to the ACBC(s) to prevent these attacks by other TRMs.

Since access control details for some devices may be quite complex, the ACBC can be simplified by off-loading some tasks onto the client processor, i.e., letting the client processor assume the more complex functions provided by a VMM. To

facilitate communication with the client processor/VMM, the ACBC can map a portion of the address space of each TRM into a distinct region of the shared primary memory (even if the TRMs are configured with built-in primary memory). Secondary storage may be divided among the TRMs and the client by adopting the mini-disk concept described earlier. The client processor can maintain the allocation information needed to simulate the mini-disks and it can load registers in the ACBC to reflect this emulation when a TRM requests mounting of a mini-disk. The client processor can translate requests and load appropriate registers in the ACBC to achieve the desired access control policy. In this fashion the ACBC design is kept simple and its checking of addresses in bus transactions can be accomplished quickly, yet a wide range of complex access control functions can be provided. This same technique can be applied to the mediation of network communication. If there is no need to monitor the access of a given TRM to the network, the ACBC can be directed to allow unlimited access and, if close monitoring is called for, the ACBC can require the TRM to forward messages through the client processor where they can be inspected and appropriately constrained.

Access to other shared resources, e.g., the terminal and other local I/O devices is generally provided on an *all-or-nothing* basis and is easily controlled by registers in the ACBC. To control access to shared primary memory, some form of mapping must be applied to TRM memory references. One or two pairs of base and bounds registers can be provided in the ACBC for each active TRM to provide mapping and access control. (In SYSTEM K there are two ACBCs, one connected to the main system I/O bus and the other to the memory bus coupler, and access control responsibilities are divided among them accordingly.) For shared resources other than primary memory, the delay imposed by an ACBC should not significantly degrade system performance due to the inherent delay in accessing those resources. In SYSTEM I and SYSTEM J the TRMs use shared primary memory only for inter-TRM communication and for service requests to the client processor, so the delay imposed by the ACBC should not seriously affect performance.

In a configuration such as **SYSTEM K**, the delay introduced by this mapping could become a problem. Moreover, the encrypted storage TRM design employed in that configuration requires cryptographic refresh of primary memory by one of the TRM SSIs. The cryptographic refresh process generates an enormous amount of bus traffic, which precludes single bus configurations for either the TRM or the main system. Even using a dual bus configuration for both the TRMs and the main system, it may be impractical to carry out the cryptographic refresh for more than one TRM simultaneously. Moreover, the refresh may effectively preclude any significant activities by the client processor due to the demands on primary memory bandwidth. Thus, in **SYSTEM K**, a TRM probably cannot execute software in a "background" mode while the client processor performs other processing. Even if a separate, shared primary memory were established solely for the use of TRMs, software in two TRMs probably could not interact for the same reason. This severely limits the utility of systems configured in this fashion.

The cost analysis discussion presented at the end of Chapter 4 suggested that one could provide 64-256 Kbytes of primary memory in the TRM (using 64 and 256-Kbit memory chips respectively) for less than the cost of hardware needed to support encrypted primary memory. Thus economic considerations also may argue for adoption of private memory TRMs in applications where primary memory size restrictions are not a problem. Private memory TRMs require only one ACBC, as opposed to the two in **SYSTEM K**, reducing system cost and further maximizing the use of off-the-shelf components. Since the single ACBC in these systems only controls access to peripherals and the shared primary memory used for inter-TRM and client processor communication, it need not exhibit extremely low delay, making it simpler and cheaper to construct. Moreover, primary memory size limitations in these TRMs may be ameliorated by use of low access time secondary storage, e.g., bubble memories, as paging/swapping devices. Thus, even though TRMs using encrypted primary memory offer greater growth potential since the

shared primary memory is readily expanded, TRMs with built-in primary memory may prove more appropriate for multi-TRM systems.

5.3.2 A Hybrid Scheme for Distributed Systems

The multi-TRM design seems especially well suited to use with proprietary software since it avoids problems of trust that arise in the third-party supplier approach. However, in the context of distributed systems, external software written by members of the user community probably cannot take advantage of the multi-TRM scheme in its pure form. First of all, it is impractical to provide at each user node a separate TRM for the external software supplied by each other user. Moreover, this scheme would not provide a basis for a distributed subsystem that includes its writer as a client! Rather, the multi-TRM approach can be used in conjunction with the third-party approach in the following fashion. Each user node can employ a multi-TRM configuration in which one of the TRMs is provided by a third-party supplier and is devoted to execution of subsystems written by members of the user community. The installation server technique described in the preceding section is employed for distribution of these subsystems. In this fashion the advantages of multi-TRM designs are available to the users but the special functionality required for secure distribution and operation of user-written subsystems is retained.

5.4 Conclusions

This chapter explored the problem of confining external software (to meet the client security requirement of preventing leakage of client information) and the related problem of supporting external software from multiple vendors in a single computer system. In developing protection mechanisms to solve these problems,

several important concepts and techniques were introduced. The two problems noted above can be unified by viewing the client as a vendor with some extra privileges that allow him to control access to shared computer system resources. Controlling access to shared resources is a major part of confining external software since network access provides the primary means of leaking client information. Two approaches to implementing multi-vendor computer systems were developed: use of a third party to supply a TRM and controlling software and use of multi-TRM computer systems.

The third-party supplier approach requires no new hardware technology; it is applicable to all of the designs developed in Chapters 3 and 4, but it does require both clients and vendors to trust third-party suppliers. A virtual machine monitor (VMM) can be used to encapsulate external software provided by various vendors (and the client) and to provide the client with a means of controlling access to system resources. The performance degradation resulting from use of a VMM should be acceptable in most application environments. A protocol based on public-key ciphers can be employed so that the third-party supplier does not have access to the external software distributed to the systems he supplies. This protocol can be enhanced so that users can acts as vendors of their own subsystems in the distributed system context.

The multi-TRM approach to confining external software supplied by one or more vendors essentially realizes a VMM design using separate processors (and, perhaps, private primary memories) for each vendor and the client. This approach minimizes the need for trusted third parties at the expense of some additional hardware: one or two access control bus couplers (ACBCs). The ACBCs filter bus transactions between the busses for the vendor TRMs and the bus(ses) of the client's processor. To keep the ACBCs simple, access control policy decisions are made by the client's processor, which loads appropriate registers in the ACBC(s) to enforce these

decisions. If the cost of the TRM-packaged components is suitably small, this approach may prove more acceptable to clients and vendors, because of the increased autonomy provided. Performance degradation associated with configurations implementing this design also should be acceptable for most applications. Moreover, such performance degradation can be restricted largely to vendor software; it should not appreciably affect client programs, due to the existence of a separate client processor and the positioning of access control hardware in the system configuration.

Chapter Six

Conclusions and Topics for Further Research

This thesis has developed and analyzed protection mechanisms for encapsulating and confining externally supplied software in personal and small business computers and certain types of distributed systems. This chapter summarizes the results of this thesis, reviewing the key concepts and techniques developed herein, evaluates the encrypted bus and encrypted storage approaches with respect to the criteria established in Chapter 1 and discusses the applicability and limitations of these approaches. The chapter concludes by suggesting some topics for further research.

6.1 Review

Chapter 1 established vendor and client security requirements associated with external software. These requirements are derived from those developed for protected subsystems in centralized computers and thus are more stringent than those that one might propose if only proprietary software were to be protected, as indicated in the review of related work. For example, other authors have not addressed the problem of detecting modification of external software (including sensitive databases constructed by the software during execution) or the problem of confining such software. The data integrity guarantee supports features such as sophisticated billing and revocation procedures for proprietary programs and is essential for many distributed system applications (see Chapter 5). These extensive, stringent security requirements yield protection mechanism designs that set this thesis apart from previous work.

In Chapter 2 the concept of tamper-resistant modules (TRMs) was explored in detail. The TRM concept is important since it embodies all of the physical protection characteristics that are a function of the level of security required in a particular environment. In this fashion none of the other protection mechanisms developed throughout the thesis need deal with physical protection issues. The monolithic-TRM design introduced in Chapter 2 illustrated some of the limitations of TRM packaging, motivating the use of cryptographic techniques to overcome these deficiencies. This design also served to introduce the secure bus coupler (SBC) in its role as a filter of transactions at the bus interface to the main TRM. The basic features of the SBC appear later in the cryptographic bus interface (CBI) and the secure storage interface (SSI) on the main TRM.

The encrypted bus approach developed in Chapter 3 introduces several important techniques in treatment of bus communication between TRMs as a special problem in communication security. The stream cipher mode developed in that chapter has been carefully designed to minimize delay and maximize throughput. In particular, this mode permits multiple crypto devices to be used in parallel to generate crypto bit stream at very high rates. The shortened DES calculation employed for CEDCs enables *simple secure* transactions to proceed at relatively high rates. Use of a distinct crypto bit stream for each simplex channel supports asynchrony in secure transaction scenarios. This is critical to the elimination of authentication checks at the slave during **simple secure read** transactions (enhancing throughput) and it allows control and data transfer connections to be combined. Finally, *aggregate secure* transactions reduce overhead on data transfers between primary memory and TRM-packaged storage devices by transmitting a cumulative CEDC at the completion of the transfer, rather than transmitting a CEDC with each transaction.

Chapter 4 employs cryptographic techniques in a fashion quite different from Chapter 3, and the encrypted storage approach introduces several important

concepts and techniques. Version tags (VTs) are employed to form version-differentiated names for cryptographically transforming storage units, and a protected version tag table (VTT) provides a basis for verifying the timeliness of storage units fetched by **Read** operations. For transfer and archival storage, the archival VTT and its associated update table provide a robust mechanism for enforcing reloading constraints for most-recent-only and non-reloadable files. The four-level hierarchic decomposition of the secondary storage VTT and appropriate caching of portions of this hierarchy make the use of encrypted secondary storage feasible. Finally, cryptographic refresh for encrypted primary memory permits the use of small VTs with cache lines, significantly reducing the amount of memory devoted security overhead.

Although Chapter 5 is short in comparison to Chapters 3 and 4, it includes several important designs (at a high level). The problems of confining external software and supporting such software from multiple vendors in a single computer system are unified by viewing the client as a vendor with some extra privileges in a multi-vendor system. The use of a TRM-based system running a third-party supplied virtual machine monitor (VMM) achieves the necessary confinement and encapsulation while minimizing the amount of trusted software. The public-key cipher protocol used in distributing external software to these computers (and in installing secure distributed subsystems) is critical to the client acceptance of the third-party approach. The multi-TRM system approach avoids the need for trusted third parties and, if economically feasible, it is probably the preferred approach. Both approaches allow the user to mediate access to the network interface, the primary means by which information can be "leaked" outside the computer.

6.2 Comparative Evaluation of the Encrypted Bus and Encrypted Storage Approaches

The primary goal of this thesis has been the design of mechanisms to protect externally supplied software in small computers. Chapter 1 established several criteria for evaluating mechanisms proposed to achieve this goal: decentralization, effectiveness, generality, flexibility, low equipment cost, minimal performance impact and transparency. The protection mechanisms developed in Chapters 3 and 4 achieve this goal in different ways and meet these criteria with varying degrees of success. Both encrypted bus and encrypted storage designs are decentralized approaches to the external software protection problem. These designs employ small computers installed at user sites and do not require any "central" computers in executing the external application software. The only time a central system might be involved is in the distribution of external software or for periodic accounting of rented/leased proprietary software.

With respect to preventing unintended exposure of information, the techniques developed in the thesis are fairly effective, i.e., if TRMs perform as specified, then only cryptanalysis or traffic analysis will yield information about the data being protected. If a suitably strong cipher is employed, then only traffic analysis remains. Neither the encrypted bus nor encrypted storage approach provides complete protection against traffic analysis, but one can limit opportunities for traffic analysis by selecting configurations that package most of the security relevant parts of the system in a single TRM. Encrypted bus designs provide greater protection against traffic analysis than corresponding encrypted storage designs since addresses in bus transactions are concealed in the former but not in the latter. For most applications, however, traffic analysis will not be viewed as a serious threat, especially at the level of T&A and secondary storage transfers. With respect to detecting malicious modification of information, the mechanisms proposed in Chapters 3 and 4 are

quite effective. An attacker has only a very small probability of circumventing these mechanisms without being detected (depending on the size of the EDC/CEDC/AICF employed).

The designs proposed in this thesis exhibit a fair degree of generality and flexibility. The protection mechanisms meet the security requirements for a wide variety of applications. Although these mechanisms have been described in the context of small computers based on a simple architecture, the general techniques developed here are applicable to a wide range of system architectures, configurations and equipment speeds. This is especially true of the encrypted storage designs for secondary and T&A storage as they are independent of most configuration and architectural details. Encrypted storage designs also offer substantial flexibility in equipment selection since they employ off-the-shelf equipment almost exclusively. Some flexibility is lost in encrypted bus designs due to possible limitations imposed by TRM packaging of non-volatile (and demountable) storage media.

Encrypted storage designs involve only one TRM and one or two SSIs whereas encrypted bus designs involve several TRMs and CBIs in most configurations. Even though encrypted storage designs waste a certain percentage of storage (that devoted to VTTs), this overhead is not likely to offset the added TRM packaging costs encountered by comparable encrypted bus designs. This is almost certainly true for systems in which secondary and T&A storage are not contained in the main TRM and is probably true when primary memory is also outside the main TRM. (This assumes TRM packaging analogous to the packaging employed for commercial cryptographic equipment.) With respect to performance, both designs introduce only a negligible delay in DMA transfers involving secondary or T&A storage not contained within the main TRM. The encrypted bus designs do hold an edge over encrypted storage designs in systems where primary memory is outside the main

TRM. (The expected increase in effective average primary access time is 0-9% for the former versus 9-18% for the latter.)

The encrypted bus approach also exhibits greater transparency than the encrypted storage approach. Aside from initialization procedures and recovery from some errors, most of the protection mechanisms are managed exclusively by the CBIs in the encrypted bus designs. In encrypted storage designs, the TRM operating system must manage VTTs for secondary and T&A storage, thus affording diminished transparency. For both approaches, applications must distinguish between files that must be protected versus those which may be stored unprotected, and the reloading constraints associated with protected files must be explicitly indicated. However, these file characteristics are obvious at the time the application is written and are easily specified as part of an operating system file creation operation.

Thus, in comparing the two approaches to protecting external software, the encrypted bus approach offers some advantages with respect to transparency, performance and susceptibility to traffic analysis whereas the encrypted storage approach provides greater generality, flexibility and reduced cost. Within a specific approach, system configuration choices offer a tradeoff of flexibility versus susceptibility to traffic analysis. Although the selection of a system design depends on requirements specific to an application environment, one can make some general observations. In both approaches, the cost of providing primary memory outside the main TRM is probably too high considering the slight gain in flexibility afforded by such configurations. When primary memory is contained in the main TRM, there is little performance difference between the two approaches. For most applications, the preferred configuration is probably an encrypted storage system with secondary and T&A storage outside the TRM. The cost, flexibility and generality advantages of this configuration probably outweigh the traffic analysis

susceptibility and the reduced transparency afforded by this configuration. This configuration is also well suited to multi-vendor, multi-TRM designs.

6.3 Applicability and Limitations

The protection mechanisms developed in this thesis have been designed for the express purpose of meeting vendor and client security requirements associated with external software in the context of personal and small business computers and certain distributed systems. The characteristics of these computer systems were established in Chapter 2. One can ask whether the protection mechanisms developed in this thesis are especially sensitive to the assumptions embodied in the system model and whether these protection mechanisms are relevant to other applications. The answers to these questions are no and yes, respectively.

The protection mechanisms developed in Chapters 3 and 4 are applicable to computer systems that do not precisely match the system model. For example, in the encrypted bus approach, the system word size and the number of bus lines employed do not critically influence the protection mechanism designs. Such differences are accommodated by changes in the amount of cryptographic bit stream generated by CBIs, but this does not significantly influence the designs, only some implementation parameters. Variations in the relative timing of the system components, including the cryptographic devices, do not seriously affect these designs although they may require minor changes, e.g. more or fewer crypto devices may be required. Substantial differences in the structure of bus transactions may require some re-engineering, but the design principles developed in Chapter 3 should still be relevant.

Most of the encrypted storage designs are even less influenced by changes in system characteristics such as word size or device timing, and these designs are

generally insensitive to details of bus operation. For secondary storage, the most critical parameter is the sector size. Changes in this parameter influence the percentage of space devoted to VTTs and EDCs but, unless the sector size changes drastically, the impact on design features such as the VTT should be negligible. Only in the case of encrypted primary memory configurations are word size, cache operation and timing details critical parameters. Here again, modifications to accommodate changes in these parameters should be possible within the context of the design principles elucidated in the chapter. Moreover, since there is only one TRM in these designs, the impact of changes in the protection mechanism details influences few components. The bottom line here is that the most promising design, SYSTEM F, is relatively insensitive to most system characteristics. In fact, since the transfer rate of many current T&A and secondary storage devices is less than 10 Mbits/s and the Fairchild DES chip set is capable of over 13 Mbits/s throughput, computer systems based on the SYSTEM F design could be constructed with current technology!

Finally, the protection mechanisms developed here can be employed for several purposes other than those described in Chapter 1. For example, one might use these mechanisms to re-enforce physical security at sites. These measures cannot prevent destruction of information stored in a computer but they can prevent disclosure and undetected modification of that information. Thus, one might purchase a TRM-packaged computer to counter these threats in environments where controlling physical access to the computer facilities is difficult or expensive to achieve. Some distributed systems employ a *file server* that provides basic file storage facilities that users can access from local nodes. The encrypted storage approach mechanisms for secondary and T&A storage can be applied by the user nodes to protect information stored at these file servers. Even some of the specialized cryptographic techniques developed in Chapter 3 may be applicable to future communications systems that exhibit very high throughput and very low

delay and which deal in very small messages. The imaginative reader may discover even more applications for these protection mechanisms.

6.4 Topics for Further Research

Several topics discussed in this thesis merit further investigation. First, the engineering of TRM packaging should be explored in depth. Details of this packaging will vary depending on the level of protection required, i.e., based on the anticipated threat environment, and there are a number of problems lurking in this area. The technology employed in existing devices such as commercial cryptographic equipment is probably appropriate for some threat environments, but both more and less elaborate packaging must be developed. An intriguing problem is the engineering of TRM packaging for a VLSI implementation of a processor, primary memory and SSI in an encrypted storage design for low to moderate security environments. Very low cost TRM packaging of this equipment might be possible if it were reduced to a just a few silicon wafers combined in a single package. (One might store keys in charge-coupled devices and rely on the inability of an attacker to disassemble the package without losing the charge on the CCD.) At the other extreme, in very high security applications, TRM packaging may have to include devices that destroy the TRM, and perhaps the would-be attacker, if tampering is detected. This type of packaging is probably unacceptable to the Consumer Products Safety Commission for home personal computers, but it may be appropriate in some military applications.

Additional work also is required in providing detailed designs for the hardware that implements the protection mechanisms developed in the thesis. For example, the functions of secure bus couplers (SBCs), cryptographic bus interfaces (CBIs), secure storage interfaces (SSIs) and access control bus couplers (ACBCs) were

described, but additional engineering design is required before a TRM-based system can be constructed using these devices. In large part these details are a function of bus characteristics, so a specific bus design must first be adopted, but other engineering questions must be resolved as well. For example, design details of bus couplers with integrated CBIs or SSIs and the ACBC must be examined with respect to buffering requirements and interaction of the control logic associated with each bus attached to the coupler. Similar design refinements are required for version tag table (VTT) management at secondary and primary storage levels. For example, the secondary storage VTT hierarchy should be tailored to the file system.

For multi-vendor computer systems there are several problems that require additional research. If a secure virtual machine monitor (VMM) is used to isolate software from different vendors and the user, then additional research is needed in the area of provably secure VMM design. Specifications of monitor calls, including those employed in inter-VM communication, must be developed if the secure VMM approach is adopted. These calls must be standardized so that vendors can produce software for execution in this virtual machine environment. If multi-vendor computer systems are constructed using multiple TRMs, vendors are relatively unconstrained in their choice of processor and memory design. However, similar standardization requirements arise with respect to communication between TRMs and the user processor operating system since that OS performs many VMM-like functions for the TRMs. Moreover, if the ACBC design is to be kept simple, it is probably necessary for TRMs to employ some standard bus interface. Thus, if multi-vendor systems are to become a reality, some standardization is required for both the VMM and multi-TRM designs.

Finally, if the protection mechanisms developed in this thesis are applied to computer systems that differ radically from those described herein, additional research will be required to work out the implementation details for these systems.

Similarly, adaptation of the protection mechanisms to applications such as the protection of information stored at distributed system file servers will require further investigation.

Appendix

Expansions of Acronyms Used in the Thesis

The following table provides expansions for acronyms used extensively in this thesis.

ACBC	access control bus coupler
AICF	authenticity/integrity check field
CBC	ciphertext block chaining
CBI	cryptographic bus interface
CC	conventional cipher
CEDC	cryptographic error detection code
CFB	cipher feedback
DES	Data Encryption Standard
ECB	electronic code book
EDC	error detection code
IV	initialization vector
PCBC	plaintext-ciphertext block chaining
PKC	public-key cipher
SBC	secure bus coupler
SSI	secure storage interface

T&A	transfer and archival
TRM	tamper-resistant module
UID	unique identifier
VMM	virtual machine monitor
VT	version tag
VTT	version tag table

References

1. Best, R. Microprocessor for Executing Enciphered Programs. U.S. Patent 4,168,396. Issued September 18, 1979.

2. Bhandarkar, D. and Juliussen, J. Semiconductor Technology: Trends and Implications. *Computer Architecture News 7*, 1 (August 1978), 4-14.

3. Branstad, D.K. Privacy and protection in operating systems. *Computer 6*, 1 (January 1973), 43-46.

4. Campbell, C. Design and Specification of Cryptographic Capabilities. Computer Security and the Data Encryption Standard, National Bureau of Standards, 1978, pp. 54-66. NBS Special Publication 500-27

5. Casey, L. and N. Shelness. A Domain Structure for Distributed Computer Systems. Proceedings Sixth Symposium on Operating Systems Principles, ACM, November, 1977, pp. 101-108.

6. Clark, D., Lampson, B. and Pier, K. The Memory System of a High-Performance Personal Computer. Xerox PARC, Palo Alto, CA.

7. d'Oliveira, C.R. An Analysis of Computer Decentralization. Technical Memo MIT/LCS/TM-90, M.I.T. Laboratory for Computer Science, October, 1977.

8. DeMillo, R., Lipton, R. and McNeil, L. Proprietary Software Protection. Foundations of Secure Computation, 1978, pp. 115-129.

9. *pdp11 Peripherals Handbook.* Digital Equipment Corporation, 1976.

10. *VAX 11/780 Hardware Handbook.* Digital Equipment Corporation, 1978.

11. Elmquist, K. *et al.*. Standard Specification for S-100 Bus Interface Devices. *Computer 12*, 7 (July 1979), 28-52.

12. Ehrsam, W.F., S.M. Matyas, C.H. Meyer and W.L. Tuchman. A cryptographic key management scheme for implementing the Data Encryption Standard. *IBM Systems Journal 17*, 2 (1978), 106-125.

13. Gold, B. *et al.*. A Security Retrofit of VM/370. Proceedings of the 1979 National Computer Conference, Vol. 48, AFIPS, 1979, pp. 335-344.

14. Hinden, H. Encryption chips sort themselves out. *Electronics 53*, 11 (June 1980), 96-97.

15. Kelly, P. Public Packet Switched Data Networks, International Plans and Standards. *Proceedings of the IEEE 66*, 11 (November 1978), 1539-1549.

16. Kent, S.T. Encryption-Based Protection Protocols for Interactive User/Computer Communication. Proceedings Fifth Data Communications Symposium, IEEE, September, 1977, pp. 5-7 - 5-13.

17. Kent, S.T. A Comparison of Some Aspects of Public-Key and Conventional Cryptosystems. ICC'79 Conference Record, IEEE, June, 1979, pp. 4.3.1-4.3.5.

18. Keys, R. and Clemens, E. Security Architecture Using Encryption. Approaches to Privacy and Security in Computer Systems: Proceedings of a Conference Held at the National Bureau of Standards, National Bureau of Standards, September, 1974, pp. 37-41. Available as NBS Special Publication 404.

19. Lampson, B.W. A Note on the Confinement Problem. *CACM 19*, 5 (May 1976), 251-265.

20. Lindsay, B. and V. Gligor. Migration and Authentication of Protected Objects. Research Report RJ-2298, IBM, August, 1978.

21. Miller, R. *et al.*. Legal Protection of Computer Software: An Industrial Survey. Harbridge House, Inc., November, 1977. Available through NTIA as PB-283 415.

22. *MC68000 16-Bit Microprocessor User's Manual (Preliminary Edition).* Motorola Semiconductor Products Inc., 1979.

23. --. Data Encryption Standard. Federal Information Processing Standards Publication 46, National Bureau of Standards, January, 1977.

24. Osborne, A. How About Low-Cost Application Software? The Answer Lies in Books. Digest of Papers COMPCON Spring 78, IEEE, 1978, pp. 46.

25. Pouzin, L. and Zimmermann, H. A Tutorial on Protocols. *Proceedings of the IEEE 66*, 11 (November 1978), 1346-1370.

26. Rivest, R.L., A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *CACM 21*, 2 (February 1978), 120-126.

27. Rivest, R., Adleman, L. and Dertouzous, M. On Databanks and Privacy Homomorphisms. Foundations of Secure Computation, 1978, pp. 169-177.

28. Rivest, R.L. performance of a prototype RSA algorithm implementation. personal communication.

29. Saltzer, J.H. and M.D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (September 1975), 1278-1308.

30. Schroeder, M. and Saltzer, J. A hardware architecture for implementing protection rings. *CACM 15*, 3 (March 1972), 157-170.

31. Schroeder, M.D. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. Th., Massachusetts Institute of Technology, September 1972. Also available as MAC TR-104.

32. Shannon, C. Communication Theory of Secrecy Systems. *Bell System Technical Journal 28*, 4 (October 1949), 656-715.

33. Svobodova, L., Liskov, B. and Clark, D. Distributed Computer Systems: Structure and Semantics. Technical Report MIT/LCS/TR-215, M.I.T. Laboratory for Computer Science, March, 1979.

Biographical Note

Stephen Thomas Kent was born in New Orleans, Louisiana, on January 25, 1951. He graduated from Ridgewood Preparatory School, Metairie, Louisiana, in 1969. He was class valedictorian, editor of the school newspaper and president of the Beta Club.

From 1969 through 1973 he attended Loyola University of New Orleans as a National Merit Scholar. As a freshman he was a recipient of an Alpha Sigma Nu honor key and was elected to Dobro Slovo, Pi Mu Epsilon and Delta Epsilon Sigma honor societies, serving as president of the last two in his senior year. Mr. Kent earned a B.S. degree *summa cum laude*, majoring in mathematics, and received the Rev. P.A. Roy Memorial Award. He was also awarded the Gold Medal for Research by the New Orleans branch of the Scientific Research Society of America for his contributions to the development of software for physical chemistry research applications.

In 1973 and 1974 Mr. Kent attended graduate school at Tulane University, studying mathematics, before becoming a graduate student in computer science at the Massachusetts Institute of Technology from 1974 through 1980. From September 1973 through June 1976 he was supported as a National Science Foundation Graduate Fellow. In June 1976 he was awarded the S.M. degree from the Department of Electrical Engineering and Computer Science and the Electrical Engineer degree in February 1977. His S.M. and E.E. thesis was entitled "Encryption-Based Protection Protocols for Secure User-Computer Communication over Physically Unsecure Channels."

From September 1977 through August 1980 Mr. Kent served as a research assistant in the Computer Systems Research Group of the M.I.T. Laboratory for Computer Science. In the summer of 1976 he worked for the Rand Corporation in Santa Monica, California, as a consultant on communication security matters. In the summers of 1977 and 1978 he worked for Bolt Beranek and Newman Inc. performing research in the area of security in computer communication networks. Since 1977 Mr. Kent has lectured in the United States and Europe on the topic of security for computer communication networks for The George Washington University, M.I.T., the University of Southern California and several private firms.

Mr. Kent is a member of the Association for Computing Machinery and its special interest groups on Operating Systems and Communications. He is also a member of the Sigma Xi scientific honorary society.

In September 1980 Mr. Kent became a member of the technical staff at Bolt Beranek and Newman Inc. He is married to Rachel Baribault Kent.

CS-TR Scanning Project
Document Control Form

Date : 8/24/95

Report # LCS-TR-259

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 254 (259-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-254) UNITED TITLE PAGE, 2-254
(255-259) SCANCONTROL, PRINTER'S NOTES, TRGT'S(3)

Scanning Agent Signoff:
Date Received: 8/24/95 Date Scanned: 8/28/95 Date Returned: 8/31/95

Scanning Agent Signature: Michael N. Cobb

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**